In-Network Techniques for Highly Reliable Datacenter Networks

Raj Joshi

National University of Singapore
2022

In-Network Techniques for Highly Reliable Datacenter Networks

Raj Joshi
(B.E.(Hons.), BITS Pilani)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2022

Supervisor:
Associate Professor Ben Leong Wing Lup

Collaborator:
Professor Chan Mun Choon

Examiners:
Assistant Professor Jialin Li
Associate Professor Richard MA Tianbai

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

Raj Joshi
December 11, 2022

Dedication

To my late mother (आई), my strength, my inspiration.

Acknowledgements

This section is quite long and I am unapologetic about it – it is just due to the sheer number of people who have been a part of this incredible journey.

I would like to express my deepest gratitude to my advisor Ben Leong for his unwavering support and guidance over the years. Throughout the course of my PhD, he has helped instill a sense of scientific rigor in me – be it while doing experiments or writing a paper. By being a staunch critic of my ideas, Ben has helped me watch out for potential pitfalls in the early stages of my projects. Through the countless challenges and hardships of my PhD journey, one of the things that has kept me going is his encouraging advice that if we work hard, do the right thing, good things will happen! Beyond research, he has also shown me what it means to be a good teacher and a good human being. My PhD has been the most challenging phase of my life so far, more due to the circumstances outside of research. Ben has went way beyond the call of his duty to support me in all possible ways, even extraditing me out of difficult situations. There have been several moments when I have felt extremely vulnerable. But during such times, I always found a sense of security in the feeling that no matter how much things go south, Ben will always have my back. I could not have asked for a better advisor and a godfather-like mentor.

I have been incredibly fortunate to have Mun Choon Chan as a collaborator, mentor and an unofficial co-advisor. His enthusiasm of working closely with his students and being hands-on with code or experiment data always made me feel that I am working "with" him rather than "under" him. He has been a staunch supporter of my ideas and has backed them since inception till paper publication, while still being critical of the problematic aspects. Faced with deepest technical issues, his timely advice has helped me find the right solutions. His ever-balanced view of things always ensured that I look at both the positive and critical aspects, and not just the critical aspects. After nearly every meeting with him, I have found more clarity of thought and felt more encouraged. I could not have asked for a better combination of advisors than what I found in both Ben and Mun Choon.

I have been fortunate to have a set of wonderful collaborators working with whom was really fun. They include Pravein Govindan Kannan, Ayush Mishra, Nishant Budhdev, Qu Ting, Boon Thau Loo (from UPenn), Chahwan Song (Mason), Xin Zhe Khooi, and Mobashir Mohammad. I also had the plea-

sure of working with and mentoring some very talented interns, notably Harsh Gondaliya and Deepanshu Jindal. Several colleagues in the lab have been very helpful with their feedback on my papers, talks or by just being a listening ear or a sounding board. Thank you Aditya Kulkarni, Oana Barbu, Zixiao Wang, Xiangyun Meng, Wang Qiang, Ebram Kamal William, Nitya Lakshmanan, and Soundarya Ramesh for being an important part of my time at graduate school. I would also like to thank Shweta Shinde for her super helpful advice, inspiration and encouragement from time to time throughout this journey.

I must also thank all the "backstage actors" who have contributed to this research and whose names have not appeared even in the acknowledgement sections of my papers (page limit constraints!). I have been fortunate to have excellent support from the admin staff of my school. They have played a crucial role in handling grants, equipment purchases, intern appointments, moving to COM3 (new building), and so many other peripheral tasks that helped support my research. Thanks are due to the entire HR, admin and building facilities teams with a special mention for Iris Chang who has been truly phenomenal in her admin support. I would also like to thank the customer support and engineering teams of `fs.com` who provided unprecedented support in supplying the research equipment I needed, sometimes doing special customizations to support my very specific requirements. Thanks are also due to Aung Nyein Kyaw (affectionately known as "bro") for running the "Cool Spot" drinks and snacks stall even on Sundays and public holidays.

I feel incredibly blessed to have the infinite support and love of my family. It is only due to my mother's many struggles and sacrifices that I am where I am today. She has been a constant source of moral support and strength for me. Destiny separated us (physically) a bit too early. But she continues to inspire me to be a fighter like her! I owe her this thesis and much more. I feel incredibly lucky to have a twin brother, Ravi Joshi, who has been unwavering in his support – right from my decision to pursue a PhD till date. Being always a bit more experienced than me in his PhD journey, he has been a constant guiding light throughout my PhD journey. I would also like to thank my cousin brother, Vinit Belwalkar, who was of immense help during the time when I juggled between taking care of my mother's cancer treatment and working on research papers – few of which received best paper awards! Thanks are also due to the newest and sweetest member of the family, my sister-in-law Tanvi Shirali, for all her love and care in the recent years. I have also been blessed to have a family friend, Anjanie Giggard, who has been keeping a motherly watch on me in the recent years.

I have had the privilege to have a wonderful bunch of friends who bring so much cheer and joy in my life. Thanks to Ashish Dandekar, Suparna Ghanvatkar, Omkar Kulkarni and Apurva Kulkarni for being my family here in Singapore. All of you have been of immense help and support during several critical phases of my PhD – be it waking me up during important deadlines or taking care of me when I have been sick. Suparna has been the central pillar of my support system in Singapore who was always there by me through all the good and bad days in the recent years. She has also been a trusted advisor for anything from inter-personal relations to statistical data analysis in Python. Akanksha Tiwari and

Tapeesh Sood have also been a great support right from the beginning of my PhD. I would also like to thank my long time friends, Kriti Aggarwal, Shivam Rai, and Kirti Bhandari for always believing in me and being there at the critical junctures of my life.

Last but not the least, I would like to thank a few people in the networking research community for their support when I needed it the most. Thank you Radhika Niranjan, Changhoon Kim, Yiting Xia, Sergey Gorinsky, Venkat Padmanabhan, and Sujata Banerjee for your support through the abysmally short personal interactions I had with you.

Contents

Abstract

Datacenters power today's large-scale Internet services such as web search, video streaming, e-commerce, and social networks for billions of users around the world. Within the datacenters, these large-scale services are realized through distributed applications running on thousands of servers which are connected by the datacenter network. The data transfers (called *flows*) between these distributed applications need to complete as quickly as possible because the flow completion time (FCT) directly impacts user experience, and thus revenue. Therefore, datacenter networks have stringent service-level requirements (SLAs) to guarantee that the worst-case (*tail*) FCTs have a tight bound. Bounding the tail FCTs in the datacenter network environment is challenging as network congestion events can cause arbitrary increase in FCTs and affect the SLAs. Further, link failures, which are a norm in datacenter networks, cause packet loss which also increases FCTs by several fold. In this thesis, we propose three in-network techniques to mitigate the increase in tail FCTs in the face of transient congestion events and link failures.

While significant prior work has been done on datacenter congestion control, in practice, complex interactions between application traffic can still lead to transient congestion events (called *microbursts*) and increase the FCTs. Mitigating microbursts requires continuous and careful tuning of system parameters based on a thorough understanding of the microbursts occurring in the network. To this end, we design and implement *BurstRadar* a system that operates in the network dataplane and monitors microbursts by efficiently capturing the telemetry information for every packet involved in microbursts. Our evaluation on a multi-gigabit testbed shows that BurstRadar incurs 10 times less data collection and processing overhead than existing solutions.

Besides transient congestion events, packet drops due to link failures also increase the tail FCT by several fold. In datacenter networks, links can fail either completely (fail-stop failure) or partially (gray failure). Both types of link failures lead to packet loss that impacts tail FCTs. Existing techniques for managing fail-stop link failures cannot completely eliminate packet loss during such failures. To this end, we propose **S**hared **Q**ueue **R**ing (SQR), an on-switch mechanism that completely eliminates packet loss during link failures by diverting the affected flows *seamlessly* to alternative paths. SQR is implemented in the network dataplane

using dataplane-programmable switches. Our evaluation on a hardware testbed shows that SQR can completely mask link failures and reduce tail FCT by up to 4 orders of magnitude for latency-sensitive flows.

A link with gray failure randomly drops packets due to bit corruption and such packet loss is significant in datacenter networks. Previous attempts to mitigate packet corruption loss seek to avoid the faulty links by routing around them, at the cost of reduced network capacities and disruption to the rest of the network. In this thesis, we investigate the feasibility and tradeoffs of the classical loss recovery strategy of link-local retransmissions in the context of datacenter networks. We present the design and implementation of *LinkGuardian*, a dataplane-based protocol that detects the packets lost due to corruption and retransmits them while preserving ordering. Our results show that for a 100G link with a loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only 9% reduction in the link's effective link speed. By detecting and eliminating tail packet losses, and avoiding timeouts, LinkGuardian improves the 99.9^{th} percentile FCT for TCP and RDMA by 18x and 160x respectively.

In summary, in this thesis, we demonstrate that it is possible for datacenter networks to perform reliably in view of transient congestion events as well as link failures using in-network techniques that leverage dataplane-programmable switches.

List of Publications

1. **Raj Joshi**, Ting Qu, Mun Choon Chan, Ben Leong and Boon Thau Loo, "BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks". Proceedings of the 9th ACM Asia-Pacific Workshop on Systems (APSys 2018). Jeju Island, South Korea. August 2018. [Chapter 3]

2. **Raj Joshi**, Ben Leong, Mun Choon Chan, "TimerTasks: Towards Time-driven Execution in Programmable Dataplanes". Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos (SIGCOMM 2019, Poster). Beijing, China. August 2019. [Chapter 5]

3. Ting Qu*, **Raj Joshi***, Mun Choon Chan, Ben Leong, Deke Guo and Zhong Liu, "SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks". Proceedings of the 27th IEEE International Conference on Network Protocols (ICNP 2019). Chicago, Illinois, USA. October 2019. [Chapter 4]
(Awarded Best Paper.)(* equal contribution)

4. **Raj Joshi**, Qi Guo, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, Ben Leong, "LinkGuardian: Mitigating the impact of packet corruption loss with link-local retransmission". Proceedings of the 6th Asia-Pacific Workshop on Networking (APNet 2022). Fuzhou, China. [Chapter 5]

5. **Raj Joshi**, Cha Hwan Song, Nishant Budhdev, Xin Zhe Khooi, Mun Choon Chan, Ben Leong, "Masking Corruption Packet Losses in Datacenter Networks with Link-local Retransmission". Under Submission (Long Paper). [Chapter 5]

List of Figures

List of Tables

Chapter 1

Introduction

Datacenters, also colloquially known as the "cloud", are central to hosting and running today's large-scale Internet services ranging from web search and e-commerce to video streaming, conferencing, and social networking. The wide-spread and ever-increasing Internet access has driven the growth of these Internet services that serve billions of users around the world today [169]. Furthermore, with the recent COVID-19 pandemic pushing more businesses and socioeconomic activities online, the global cloud computing market size is expected to grow to USD 947.3 billion by 2026 [124]. For ensuring scalability and reliability, these large-scale Internet services are often implemented as distributed applications that run across a large number of servers connected by a network [96]. Consequently, today's datacenters are massive computing infrastructures with hundreds of thousands of servers interconnected by a large and high-speed network with peta-bit scale bandwidth at its core [164].

Majority of today's Internet services that are hosted and run by the datacenters are *soft real-time* [8]. This is because they are latency-sensitive and the failure to meet the response deadline adversely impacts user experience and thus revenue [8, 34, 50, 85, 157, 168]. For example, experiments at Amazon showed that increased latency in page load times led to decrease in online sales while at Google increase in search results display

time led to decrease in revenue [113]. A study by Akamai shows that a delay of 100 ms in website load time can hurt the conversion rate[1] by 7% [4]. Besides directly affecting the revenue, Internet service providers, in certain situations, also suffer additional losses in terms of compensation to customers when the performance and availability guarantees are not met [170].

The total permissible latency for an Internet service is determined by customer impact factors of the specific service [114]. After excluding the Internet and the client rendering delays, what remains is the "backend latency" limit which needs to be met by the distributed applications running in the datacenter [8]. This backend latency budget gets further divided into server processing and network communication stages required by the distributed applications implementing the Internet service [96, 173]. The typical request/query processing workflow of an Internet service consists of many sequential stages involving parallelization across 1000s of servers and aggregation of responses across the network [96]. For example, the processing workflow for a Bing search query on average involves 15 stages where 10% of stages process the query in parallel on 1000s of servers [96]. Similarly, a popular page load request on Facebook can require fetching 1000s of distinct objects distributed across 100s of memcached servers [142]. As the processing workflows involve 1000s network communications (flows), 10s of them occurring serially, the latency budget for each individual flow is on the order of microseconds [9, 20]. In this way, the user-level service deadlines ultimately translate into flow completion time (FCT) targets for the network communication between the distributed applications within the datacenter [180]. As a result, the *tail* (worst-case) flow completion times (FCTs) of datacenter network communication have a direct impact on the user-level service deadlines and thus the revenue [51]. Consequently, datacenter network operators are required to provide stringent SLA guarantees on tail FCTs at a microsecond scale [8, 51, 68, 142, 155]. For example, even if the SLA guarantees 99.9% of the

[1] percentage of website visitors that take a desired action.

flows to complete within a bounded time, for processing requests involving 100 flows, the probability that at least one of the 100 flows will face higher FCT and affect the overall request processing is ∼9.5%.

We note that apart from the flow completion times of network data transfers (flows), the tail performance of request processing in datacenters is also affected by the tail performance of the involved server processing [96]. There is a separate body of literature for addressing tail performance due to server processing [52, 118, 126]. However, in this thesis, we focus on the network-related sources of tail performance i.e. the FCTs for the network flows.

In the following subsection, we describe the challenges involved in providing SLA guarantees on tail FCTs in today's datacenter networks.

1.1 Challenges in providing tail FCT SLA Guarantees

Providing a tight bound on tail FCTs in today's datacenter networks is challenging as several factors can lead to abnormal increase in the FCTs and violate the SLAs. Broadly, the various factors can be categorized into two main categories - congestion events and failure events. We elaborate on them below.

1.1.1 FCT increase due to congestion events

Several links in a datacenter network are shared across 1000s of servers. Depending on the traffic patterns, these links can become congested (at short timescales) causing packet queues to build up on switches. Such queue build ups cause transmission delays and increase the tail (worst-case) FCTs. It is, therefore, not surprising that a significant amount of prior work has been done on datacenter congestion control [8, 9, 10, 17, 76, 83, 87, 149, 173, 180] and load balancing [7, 109] to keep the queue occupancies low and thereby prevent increase in the tail FCTs. Yet, unpredictable congestion events

(microbursts) can still occur due to different operating conditions (compared to the original design) and due to complex unanticipated interactions of different flows. We elaborate on these below.

Many of these prior designs are based on the assumptions that may not hold in practice. For example, although DCTCP was designed to keep the queue occupancies low [8], Judd [101] reported that in their deployment of DCTCP in Morgan Stanley datacenters, it did not keep the queue occupancies low under certain conditions and required further tuning based on the link speeds and traffic characteristics. Also, application traffic patterns evolve over time which can invalidate some of the previous assumptions. For example, for a web search workload, less than 1% flows were reported to have flow sizes less than 1000 bytes in 2010 [8] compared to 95% flows in 2018 [134]. Also, much of the prior work focuses on preventing "systematic" congestion events under assumed operating conditions and therefore cannot prevent the unpredictable congestion events that arise from complex unanticipated interactions between flows. For example, Google's datacenter network fabric supports thousands of distinct applications and services [164], each with different traffic characteristics. Furthermore, different applications can employ different congestion control algorithms, especially in multi-tenant datacenters. Due to different application requirements, having a mix of congestion control is possible even when the datacenter is under a single administrative control [101]. This leads to complex and unanticipated interactions between flows as there is no precise control over how independent flows interact with each other at the network switches along their paths [97]. Other factors leading to microbursts include TCP incast in partition-aggregate traffic patterns [8, 115], occasional synchronization of application traffic [105], TCP segment offloading or application-level batching [160]. Popular webservices such as LinkedIn have reported the occurrence of microbursts leading to increased network latency [103]. Measurements from Facebook datacenter estimate that microbursts can occur as frequently as 200 μs and last 100's of μs [188]. In today's datacenter networks, since the normal

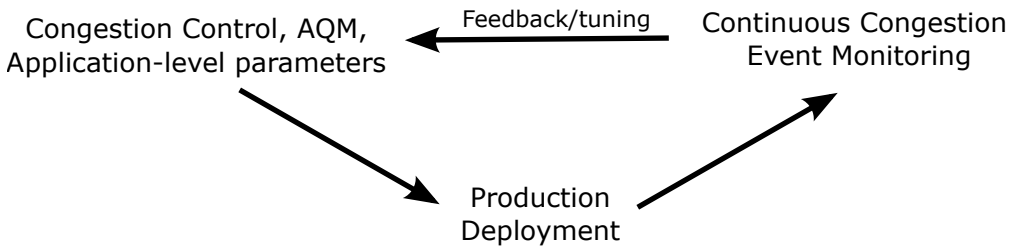


Figure 1.1: Lifecycle of managing unpredictable congestion events in datacenter networks.

end-to-end network delays are on the order of 10's of μs [60], an occasional 100 μs queuing delay becomes unacceptable and affects the SLAs [189].

Overall, it not plausible to design congestion control, load balancing and active queue management (AQMs) schemes for a datacenter network that take into account *all possible* complex interactions and work together to keep the tail FCTs low. In other words, handling congestion events with the management strategy of "design, deploy and forget" is not possible since unpredictable congestion events are likely to occur due to unanticipated interactions. Instead there is a need to continuously monitor unpredictable congestion events, find their causes, and use this information to mitigate future occurrences by adapting the deployed congestion control, AQMs, load-balancing, etc. Figure 1.1 shows this management strategy where, post deployment, continuous monitoring of congestion events provides the feedback required to fine-tune and adjust the deployed schemes such that the current causes of unpredictable congestion events are eliminated.

1.1.2 FCT increase due to network failure events

Another major cause of increase in FCT is network failure events that typically cause packet loss. In this thesis, we focus on network link failures as they are more prevalent compared to network device failures [73]. Network link failures are of two types: (i) **Fail-stop link failures:** when the network link connection between two network devices is completely "down". (ii) **Gray link failures:** when the network link connection

between two network devices is "up", but the link corrupts certain packets. Both types of link failures result in packet loss. The packet loss could be transient if the network management system is able to detect the link failures and route traffic such that it avoids the failed link. However, for short-latency sensitive flows, even a small amount of packet loss can increase the FCT by several fold [152]. Below we provide further details on the two types of link failures.

Fail-stop link failures. Fail-stop link failures occur when one or more of the hardware components forming the link fail. These components include line cards, transceivers, fiber/copper cables, etc. They can also be caused by connection problems due to carrier signaling/timing issues [62]. Datacenter networks typically use commodity hardware in order to trade-off significant hardware costs for slightly reduced reliability [21, 193]. While the individual network components have a small but non-zero failure rate, with thousands of switches and tens of thousands of links in a datacenter, the aggregate link failure rate across the datacenter network can be significant enough to inflict sufficient packet loss and affect the SLAs. In practical datacenter operating conditions, the reported MTBF (Mean Time Between Failures) for network links is on the order of 10,000 hrs [127]. Using this, we can estimate the hourly link failure rate as below,

$$Failure\,Rate = \frac{1}{MTBF} \tag{1.1}$$

For a MTBF on the order of 10,000 hrs, the failure rate comes to be $\sim 10^{-4}$ per hour. Since the number of links in a large datacenter network are on the order of $\sim 10^{5}$, a failure rate of 10^{-4} per hour means that we can expect 10's of links to fail every hour. This example calculation indeed corroborates with real-world data from production datacenter networks. Gill et al. reported an average of 40.8 links failing each day [73]. At the 95^{th} percentile, about 136 links are reported to fail daily [73].

Gray link failures. Compared to fail-stop link failures, gray link failures have very

different failure characteristics. Gray link failures typically occur on optical links. In datacenter networks, switch-to-switch links are typically optical [182, 192] as they can support high link speeds (10-400 Gbps) over long distances compared to electrical links. Optical links are susceptible to packet corruption, as the optical receiver sometimes fails to correctly decode the transmitted bits. Optical decoding errors can occur due to a variety of reasons such as fiber bending, connector or fiber tip contamination by airborne dirt particles, decaying laser transmitters, etc. [182, 192]. When an optical decoder decodes the bits of a packet incorrectly, the Ethernet frame checksum (FCS) fails and the receiving MAC drops the packet. With tens of thousands of optical links in a datacenter, packet corruption loss can be significant. A large-scale study by Microsoft consisting of 350K links across 15 datacenters shows that the number of packet lost due to corruption can be on par with the number of packets lost due to congestion [192]. Another study by AliBaba showed that about 18% of the packet drops that caused serious network performance degradation to their cloud customers were caused due to corruption [189].

Overall, packet loss due to both types of link failures is a norm in datacenter networks. It therefore needs to be handled for ensuring reliable delivery of packets such that end-to-end retransmissions and retransmission timeouts do not occur and thus the tail FCTs remain bounded. One way to prevent packet loss due to link failures could be to use highly reliable hardware. However, highly reliable network hardware is prohibitively expensive, especially for the large scale of the datacenter networks. The challenge therefore lies in achieving reliable (nearly) no loss packet delivery on top of commodity networking hardware. Other aspects of datacenter computing such as datacenter storage have long followed the trend of using cheaper commodity hardware (e.g. disks) and masking the hardware unreliability from applications through intelligent techniques [71]. Similar masking of unreliability needs to be achieved for datacenter networking hardware.

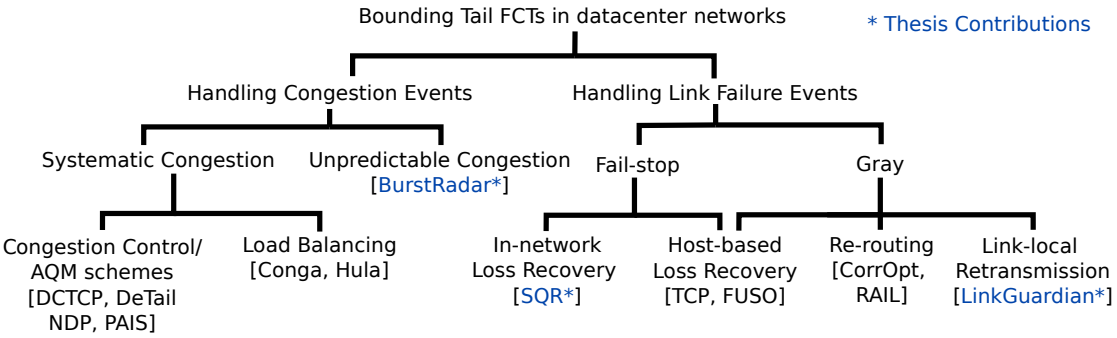


Figure 1.2: Problem space for bounding tail FCTs in datacenter networks

1.2 The In-Network Approach

Figure 1.2 shows the problem space for ensuring bounded tail FCTs in datacenter networks. A vast majority of the current solutions to handle congestion events as well as link failure events take the *end-to-end approach* i.e. they treat the network as a black box that simply forwards packets and solutions run mainly on the end host servers. The end-to-end approach works really well for handling systematic congestion events as their causes are mainly rooted at the hosts. As a result, systematic congestion is a well-researched problem with decade-worth of prior work (discussed in §1.1.1). However, as we show through the following chapters, the end-to-end approach is insufficient to mitigate the impact of unpredictable congestion events or handle link failure events in order to ensure bounded tail FCTs.

In this thesis, we take an in-network approach to mitigate unpredictable congestion events as well as handle link failures. In our in-network approach, the solutions run entirely within the network while being nearly transparent to the end hosts. The key enabler for our in-network approach is a new hardware technology called the *dataplane programmable switches* [36, 95, 139]. These switches allow us to implement algorithms within the forwarding chip (dataplane) of network switches. Once implemented, these algorithms then run at hardware speeds ($\sim$ Tbps). Below we list the key new function-

alities enabled by programmable switches which make our in-network approach feasible:

- **Packet-level operation:** Even at terabit-level aggregate speeds, programmable switches allow us to perform operations at a per-packet granularity i.e. they allow us to implement algorithms at the highest resolution of a single packet.
- **Stateful operation:** Programmable switches provide state in the dataplane that can be accessed at hardware speeds (∼Tbps). This allows implementing stateful algorithms and protocols where the logic spans several packets.
- **Packet cloning:** These switches allow creating copies of the packets at hardware speeds. They also provide different on-chip paths that allow us to move these packet copies and place them in any output (egress) queue. This essentially enables the possibility of performing switch-based packet retransmission at hardware speeds.
- **Per-packet queuing telemetry and precise timestamping:** Programmable switches are also able to track queue size information on a per-packet basis. For example, for each packet, the hardware can provide information about the queue size when the packet was enqueued and the queue size when the packet was dequeued from the queue. Within the dataplane, switches also provide per-packet timestamping at a nanosecond resolution [106]. The per-packet queuing telemetry combined with precise timestamping provides high-resolution data to reconstruct any unpredictable congestion events that last only 100's of μs.
- **On-chip packet generation:** Programmable switches provide different ways to generate new packets at hardware speeds. This makes it possible to implement protocol messages between different switches at hardware speeds and also allows to transfer any telemetry information out of the switch dataplane (switching chip).

Armed with these new functionalities enabled by programmable switches, in this the-

sis we propose three in-network solutions with a common goal of ensuring bounded tail FCTs in datacenter networks. Specifically, for unpredictable congestion (microbursts), we propose BurstRadar which provides continuous and efficient in-network monitoring of microbursts irrespective of the cause. BurstRadar helps the network operators to identify the cause for every single microburst occurring in the network so that the network operators can take corrective actions such as tuning application and congestion control parameters, addressing an offending flow/application, etc. Basically, BurstRadar provides continuous congestion event monitoring which is an integral part of managing unpredictable congestion events (see Figure 1.1). For fail-stop link failures, we propose SQR which performs *seamless* in-network packet loss recovery such that the end-host applications can remain completely oblivious to any link failures occurring in the network. SQR runs locally on a single switch and requires no coordination with other switches since fail-stop link failures can be detected locally. For handling gray link failures, we propose LinkGuardian which also performs in-network packet loss recovery. While LinkGuardian uses similar techniques as SQR for packet cloning and retransmission, the key difference is that LinkGuardian needs to perform selective retransmission of only the packets that were lost due to corruption packet loss. To do so, LinkGuardian implements a fast and efficient link-local packet loss detection and retransmission protocol that runs between two adjacent switches that share the corrupting link.

1.2.1 Why the In-network approach works?

As discussed in §1.1.1, continuous monitoring of congestion events is required to find the current causes of unpredictable congestion events so that they could be fixed later. However, due to the transient nature of the congestion events (lasting 100's of μs), an end-to-end approach [14, 135] fails to even detect the congestion events, let alone collect sufficient telemetry information that enables finding the root cause. The key insight behind BurstRadar's in-network approach is to capture the unpredictable congestion

events locally within a switch's dataplane (where they occur) and then export the per-packet telemetry information for each congestion event. This is enabled by the per-packet queuing telemetry, precise timestamping, stateful operation and on-chip packet generation functionalities of programmable switches.

In case of link failures, the resulting packet loss leads to the increase in the flow completion times (FCTs). The key reason for this is that, by default, the packet loss detection as well as recovery is performed in an end-to-end manner. With an end-to-end approach, the packet loss recovery incurs a delay of at least 1 round-trip time (RTT). Also, since an end-to-end approach typically uses a sequence number based scheme to detect packet loss, it fails to quickly detect the loss of the last (tail) packet of a flow and relies on an expensive retransmission timeout (RTO) which significantly impacts the FCT. Furthermore, an end-to-end approach cannot distinguish between a corruption and a link failure packet loss. As a result, in an end-to-end approach, any packet loss gets treated as a congestion loss leading to reduction in the sending rate of the transport-level sender and thereby increasing the FCT. The key insight behind the in-network approach adopted by both SQR and LinkGuardian is to mask the packet loss due to link failure events from the end-host transport by performing in-network packet loss recovery. By doing so, we can prevent the increase in FCT due to link failures by avoiding all the above drawbacks of an end-to-end recovery. By operating the detection scheme locally at a network link, SQR and LinkGuardian are able to precisely and quickly (at hardware speeds) detect the packet loss due to link failure events without requiring an expensive RTO. Also, since the packet loss recovery is performed in-network and at hardware speeds, the recovery delay is less than 1 RTT. This in-network packet loss detection and recovery is enabled by precise timestamping, packet cloning, stateful operation and on-chip packet generation functionalities of programmable switches.

1.3 Summary of Thesis Contributions

In the following subsections, we provide a brief overview of our contributions.

1.3.1 BurstRadar

As discussed in §1.1.1 unpredictable congestion events (microbursts) can affect FCTs in datacenter networks by causing increased latency, jitter and packet loss. To address this problem, we first need to be able to accurately detect the occurrence of these microbursts and identify the contributing flows. However, it is hard to do so since microbursts occur unpredictably and last only for 10's or 100's of μs. This is further exacerbated by the fact that, when microbursts occur, the telemetry information needs to be captured at full link speeds while the link speeds in modern datacenter networks are ever-increasing (up to 800 Gbps as of today [78]).

Our system, called *BurstRadar*, is designed to run entirely in the switch dataplane. Higher link speeds are correspondingly supported by faster switch dataplanes and therefore BurstRadar's design and implementation is agnostic to the link speeds. Further, for efficiently capturing the unpredictable microbursts, our key insight is that microbursts are localized to a port's egress queue. This makes all the information required for detecting and characterizing a microburst available *together* on a single switch. Unlike the in-band network telemetry approach [77, 98], by detecting a microburst directly on the switch where it happens, BurstRadar can avoid the computations and delays arising from having to correlate monitoring information from different points in the network. BurstRadar uses a *Snapshot algorithm* to capture information of the packets involved in a microburst. It then generates on-demand *courier* packets for transporting this information together.

We have implemented BurstRadar on an Intel Tofino [139] switch and evaluated it on a multi-gigabit hardware testbed using utilization burst distributions from Facebook's

production network [188]. Our results show that even with microbursts occurring as frequently as every 200 μs, BurstRadar processes 10 times less telemetry information compared to existing solutions [77, 111], while providing all information to fully characterize microbursts and identify the contributing flows. BurstRadar captures telemetry information for all packets contributing to microbursts, even with bursts occurring simultaneously on multiple egress ports, while consuming very few resources in the switch dataplane.

Through the design and implementation of BurstRadar, we demonstrate that programmable dataplanes can be used to detect microbursts more efficiently by capturing the telemetry information of only the packets involved in microbursts.

1.3.2 SQR

As discussed in §1.1.2, fail-stop link failures cause packet loss which can increases the FCTs. Existing management techniques for fail-stop link failure involve detecting a link failure and redirecting traffic on an alternative backup path. However, these techniques cannot keep the tail FCTs low under link failures because they cannot *completely* eliminate packet loss during the failure detection and route reconfiguration. As a result, loss recovery is required to be done by end-hosts which can increase the FCTs by several fold. We observe that to completely mask the effect of packet loss and the resulting long recovery delay, the network has to be responsible for packet loss recovery, instead of relying on end-to-end recovery. Our system, called **S**hared **Q**ueue **R**ing (SQR), is an on-switch mechanism that completely eliminates packet loss during link failures by diverting the affected flows *seamlessly* to alternative paths. In SQR, our key idea is that by estimating the upper bound on the link failure detection and the network reconfiguration delay, the switch dataplane can *cache* a copy of the recently sent packets for this duration. Then, in the event of a link failure, we can avoid packet loss by retransmitting the cached copy of these previously transmitted packets on the alternative backup

path. We have implemented SQR on an Intel Tofino switch using the P4 programming language. Our evaluation on a hardware testbed shows that SQR can completely mask link failures and reduce tail FCT by up to 4 orders of magnitude for latency-sensitive workloads.

While caching packets on the switch is an obvious idea, it is not straightforward to achieve and was not feasible until now. The significant reduction in route recovery times and increase in on-switch packet buffer sizes have made it feasible, while our design, implementation and evaluation of SQR demonstrates that it is both effective and practical. Our work suggests that on-switch packet caching would be a useful primitive for future switch ASICs.

1.3.3 LinkGuardian

While SQR helps eliminate packet loss during fail-stop link failures, it does not help with gray link failures since the link remains "up" but still drops packets due to corruption. Packet corruption loss is a serious problem in datacenter networks. A large-scale study by Microsoft reported that the number of packets lost due to corruption is comparable to those lost due to congestion [192]. Packet corruption loss is different than congestion packet loss because it does not go away even when the end hosts reduce their transmission rates. Unless mitigated, packet corruption will continue to cause degradation to application performance and affect a cloud provider's SLAs (Service Level Agreements) [189] by impacting both latency-sensitive and throughput-sensitive applications.

Previous attempts to mitigate the impact of packet corruption loss seek to avoid the faulty links by routing around them, at the cost of reduced link capacities and disruption to the rest of the network. In this thesis, we investigate the feasibility and tradeoffs of the classical loss recovery strategy of link-local retransmissions in the context of datacenter networks. Through the design and implementation of LinkGuardian, we show that it is feasible to perform ordered link-local recovery of corruption packet loss on

today's high-speed datacenter links thereby making the end hosts completely oblivious to corruption packet loss. LinkGuardian is designed as a dataplane-based protocol between two neighboring switches that are connected by a link with gray failure. The sending switch makes a copy of the recently sent packets and buffers them for potential future retransmission. The two switches run a protocol in the dataplane to detect corruption packet loss, if any. In case of a loss, the sending switch retransmits the lost packet. The receiving switch in the meantime buffers the out-of-order packets and transmits them ahead "in order" once it receives the retransmitted copy of the lost packet. Since flows in today's datacenter networks are mostly short and use TCP, we found that even out-of-order retransmission by LinkGuardian can be effective in mitigating the impact of corruption packet loss.

LinkGuardian is implemented using dataplane-programmable switches and is amenable to incremental deployment. For deployment, we propose a combined LinkGuardian + CorrOpt [192] solution to efficiently manage link corruption in large-scale modern datacenter networks. CorrOpt [192] is the state-of-the-art solution that disables corrupting links subject to network capacity constraints.

Our evaluation on a hardware testbed shows that: (i) For a 100G link with a loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only a 9% reduction in the link's effective link speed; and (ii) LinkGuardian improves the 99.9^{th} percentile FCT for TCP and RDMA by 18x and 160x respectively by handling tail packet losses at sub-RTT timescales. Using large-scale simulations, we also compared the combined LinkGuardian + CorrOpt [192] solution with vanilla CorrOpt. Our results show that the combined solution reduces the total network-wide packet loss rate by at least 4 orders of magnitude and also allows network operators to operate at higher capacity constraints which were not possible before.

Overall, we believe that we have made a strong case that link-local retransmission is both practical and effective for modern datacenter networks.

1.4 Thesis Structure

The rest of this thesis is structured as follows. Related work is reviewed in Chapter 2. We present the background, motivation, design and evaluation of BurstRadar and SQR in Chapters 3 and 4 respectively. In Chapter 5, we cover LinkGuardian in full details. Finally, in Chapter 6, we discuss the future directions and conclude this thesis.

Chapter 2

Related Work

As described in Section 1.1, the two main causes for increase in tail FCTs include congestion events and link failure events. In this chapter, we therefore review the related work in these two broad categories.

2.1 Handling Congestion Events

As shown in Figure 1.2, congestion events that lead to increase in tail FCTs can be of two types - (i) systematic congestion events that are caused by the design of end-host congestion control, on-switch AQMs, and/or load-balancing; and (ii) transient congestion events that are caused by unanticipated complex interactions between the flows in the network.

There is a large body of work towards addressing high tail FCTs due to systematic congestion events. This is a well understood area and here we provide a high-level summary of the most relevant works. DCTCP [8] and HULL [9] propose improvements to TCP in order to reduce queue occupancy in datacenter networks. D^3 [180], D^2TCP [173], and PDQ [87] belong to the category of protocols that take into account end-host specified deadlines for completion of the flows. PIAS [17], QJUMP [76] and pFabric [10] use

strict priority queue scheduling in combination with per-flow priority specified by end-host applications. pFabric also requires an unconventional AQM where a high priority incoming packet could replace a low priority packet in the switch buffer. pHost [69], NDP [83], Homa [134] and ExpressPass [43] use receiver or credit-based scheduling and priorities. HPCC [119], DCQCN [190], XCP [108] and RCP [54] use an explicit congestion feedback. Timely [131] use one-way delay as a congestion signal. Swift [115] handles congestion both in the network as well as at the end host. Load balancing schemes such as CONGA [7] and HULA [109] also help avoid systematic congestion and keep the tail FCTs bounded. AQMs such as Approximate Fair Queuing [161] help to prevent long flows from increasing the FCTs of short flows. However, scheduling itself does not reduce the *overall* occupancy of the shared buffers on the switches. Buffers can still fill and cause packet loss thereby affecting the FCTs [74].

Almost all of the above proposals operate under certain assumptions about the operating environment and therefore can handle any systematic congestion events. However, as described in Section 1.1.1, despite of the significant prior work, microbursts can still occur when the design assumptions of these schemes do not hold in practice or when there are complex unanticipated interactions between the flows. Therefore, what is not addressed well in the literature is the increase in tail FCTs due to non-systematic transient congestion events i.e. microbursts. As explained in Section 1.1.1, it is not possible to completely *prevent* the occurrence of microbursts as they occur due to complex unanticipated interactions of the different flows in the network. What is therefore needed is a feedback loop as shown in Figure 1.1. Any transient congestion events occurring in the network not only need to be detected, but the relevant telemetry information also needs to be collected so as to identify the exact cause of the microburst. This information is necessary to take appropriate corrective action and prevent future occurrence of microbursts due to the same cause in the future. In the following subsection, we cover in details the existing literature on detecting microbursts and collecting the relevant

telemetry information.

2.1.1 Monitoring Microbursts

Commercial solutions such as Cisco's Nexus 5600 and 6000 series switches, as well as Arista's 7150S series switches can detect the occurrence of microbursts but provide no details about the cause [45, 138]. Learning the cause requires traffic mirroring and data correlation across different monitoring data streams [45]. In contrast, BurstRadar provides a full snapshot of telemetry information about the packets involved in a microburst. With this information, we can identify the contributing flow(s) without the significant costs associated with data correlation and traffic mirroring. Marple [137] is another network monitoring system that proposes augmenting dataplane programmability with a custom key-value store hardware primitive. It presents a microburst detection case study in which microbursts are assumed to occur at regular intervals. The proposed approach will not work in practice because microbursts do not occur at regular intervals [188]. BurstRadar does not make any such assumption about the arrival pattern of microbursts. It may be possible to orchestrate BurstRadar's techniques in the Marple framework with some modifications. However, unlike BurstRadar, the hardware primitives required by Marple are not available on today's programmable switching ASICs.

In-band Telemetry (INT). In-band Telemetry (INT) [77, 111] is a network debugging system that is built on top of programmable dataplanes. It is the state-of-the-art solution that can be deployed to monitor microbursts or non-systematic transient congestion events. Later in Chapter 3, we compare our solution BurstRadar with INT. Figure 2.1, shows an example operation of INT. Consider flow 1 in Figure 2.1 that traverses switches 1, 3, 4, and 5. If INT is enabled for flow 1, then when a packet from flow 1 (P1 in Figure 2.1) enters the network, the first switch called the "INT Source" (switch 1) adds an INT header into the packet (see green header 1 attached to P1 in Figure 2.1). This telemetry header consists of information such as the timestamps from

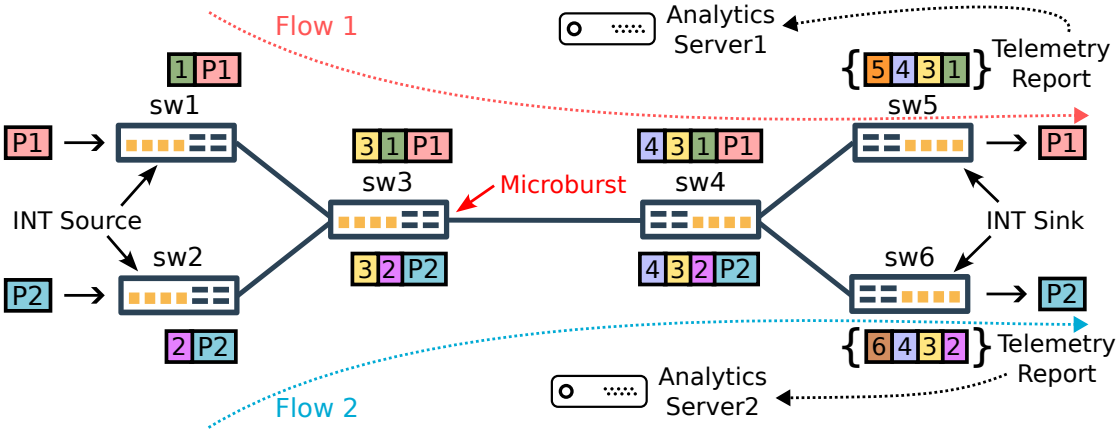


Figure 2.1: Example operation of In-band Network Telemetry (INT) to monitor microbursts

switch 1 corresponding to when P1 arrived and left the switch, the enqueue/dequeue queue depth (congestion) experienced by P1, etc. All the subsequent switches add similar information to P1. Finally, when P1 is processed by the last switch called the "INT Sink", all the INT telemetry headers are removed from the packet and sent to an analytics server for processing. The packet then exits the network in the same form as it entered the network without any INT headers. When deployed across a large datacenter network, the INT Sink switches export telemetry data to multiple different analytics servers for load balancing. Now consider a scenario, where a burst flow (flow 2) enters the network and causes a transient congestion event (microburst) at switch 3. Now, to use INT to detect that a microburst occurred at switch 3 and it occurred due to the bursty flow 2, we would also need INT to be enabled on flow 2. Thereafter, we would need to correlate the INT telemetry information from both the flows in the following manner. When packets from flow 1 show that they experienced high queuing delay on switch 3, we would extract the switch 3 timestamps from these specific packets. Then we would check the INT information inside packets from other flows (in this case flow 2) to see if any packets from other flows were at switch 3 around the same time as the packets from flow 1. In this case, we would find that the packets from flow 2 (processed

at analytics server 2) were at switch 3 and were the cause for the microburst.

While Figure 2.1 showed a simple example, in practice, however, due to the complex interactions of datacenter network traffic, microbursts are unpredictable [188] and can occur at any time involving any flows. What this means is that, in order to detect and characterize microbursts reliably, INT would need to be enabled for all flows at all times. The INT analytics servers would then need to process telemetry information for every single packet in the network, even though only a small number of these packets are involved in the microbursts. Further, as we saw in the example above, expensive data correlation across analytics servers would then be required to reconstruct a microburst event. Furthermore, enabling INT on all flows would consume 10% additional bandwidth[1] in the entire network due to the extra INT headers. BurstRadar, on the other hand, captures telemetry information only for the packets involved in microbursts, does not require expensive data correlation and is non-intrusive to production traffic since it operates out-of-band.

Chen et al. recently proposed Snappy, a technique to estimate the contents of a microburst queue and identify the culprit (heavy) flows in the dataplane [39]. Snappy's detection of culprit flows is however probabilistic in nature and the probability of identifying all the culprit flows (*Recall*) increases with the number of switch pipeline stages used by Snappy. Snappy is expected to require more than 128 stages for achieving a decent "recall" in practice[2]. Today's programmable switching ASICs do not currently support such a large number of pipeline stages and we believe that they are unlikely to be available in the near future due to cost concerns. Snappy further requires division and rounding operations which are not currently supported on high-speed programmable switching ASICs. BurstRadar, on the other hand, requires only a modest amount of resources (§3.3.3) and can thus be implemented on programmable switches available

[1]For a 5-hop diameter network, INT requires extra 54 bytes per packet [98] which is 10% extra for a median packet size of 500 bytes [23].

[2]Real-world microbursts queue lengths are less than 250 KB at the 90th percentile [188]. This requires Snappy to use smaller "window" sizes to better approximate the queue boundaries.

today. The microburst information exported by BurstRadar goes beyond accurate culprit detection and provides a full characterization of microbursts, which is important to network operators for network planning. While Snappy's approach of detecting culprit flows in the dataplane is more suited for automatic microburst mitigation, further work is required to make it practical.

NetSight [82] employs mirroring or packet cloning for exporting telemetry information for *every single* packet traversing a switch. Since packets involved in microbursts form a very small fraction of the overall packets, such an approach is grossly inefficient and infeasible for large datacenter networks. Everflow [191] uses "match and mirror" to selectively trace *specific* packets across a large datacenter network. However, since microbursts are unpredictable [188], identifying the specific packets involved in a microburst and exporting the corresponding queuing information requires going beyond the stateless "match and mirror" operation.

There have been systems proposed for monitoring a different class of microbursts, called *link utilization* microbursts [171, 188]. Link utilization microbursts are intervals where the utilization of a link exceeds a certain threshold, and unlike queuing microbursts, might not result in queuing. These systems [171, 188] can only detect *link utilization* microbursts at the time-scale of tens of microseconds. BurstRadar instead monitors *queuing* microbursts at a sub-microsecond resolution. It remains a future work to extend BurstRadar to monitor *link utilization* microbursts.

2.2 Handling Link Failure Events

As shown in Figure 1.2, other than the congestion events, a major cause for the increase in tail FCTs are the link failure events. Link failure events are broadly of two types – fail-stop and gray – and in this section we review the literature for both of them.

2.2.1 Handling Fail-stop Link Failures

The related work for handling fail-stop link failures can be further categorized into two categories – **(i) Route Recovery:** this includes the works that find an alternative route for traffic that was previously carried by the failed link, and **(ii) Packet Loss Recovery:** which includes the works related to recovering the packets that were lost from the time the original link failed and the new alternative route was established.

Route Recovery. Among existing route recovery schemes, many attempt to achieve *fast re-routing* for multi-path datacenter topologies. Failure carrying packets [116] are proposed to avoid route convergence delay by carrying failed link(s) information inside data packets to notify other nodes. Fast Reroute (FRR) [147] used in MPLS networks can provide recovery in less than 50 ms during a link/node failure. Packet Re-cycling [123] takes advantage of cycle in the network topology where routers implement a cyclic routing table. SPIDER [35] and Blink [86] maintain a pre-computed backup next hop in the switch. Sedar et al. [158] implement the fast reroute primitive based on known port status in programmable data planes and in Data-Driven Connectivity [121] dataplane packets are used to ensure routing connectivity. Flowlet switching [104] based load balancing schemes such as CONGA [7] and HULA [109] are an implicit form of fast re-routing schemes since they avoid a failed path for routing subsequent flowlets. Another group of route recovery schemes consist of multi-path network architectures that allow fault-tolerance [3, 5, 75, 79, 81, 165]. Notably, F10 [122] designs an AB fat-tree and a centralized rerouting protocol to support downlink recovery.

ShareBackup: ShareBackup [181] is the state-of-the-art solution for route recovery. Later in Chapter 4, we evaluate our proposed solution SQR in combination with ShareBackup. The basic idea of ShareBackup is to have a shared pool of backup switches spread across the entire datacenter network. These backup switches essentially form a backup network that provides on-demand alternative paths when existing paths fail due

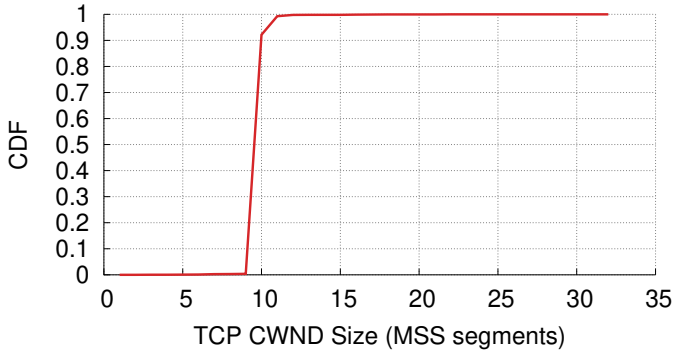


Figure 2.2: `cwnd` size distribution for short flows

to fail-stop link failures. ShareBackup uses optical circuit switches which can dynamically form new backup links on-demand to provide the required backup paths. Whenever a link faces a fail-stop link failure, ShareBackup reconfigures its pool of optical circuit switches such that a new alternative link is formed between the two switches who lost the link between them. ShareBackup takes only about 730 μs to reconfigure a backup link making it the state-of-the-art for route recovery. The main drawback of ShareBackup is that the delay of 730 μs is still not short enough to prevent increase in FCTs for short flows in today's high-speed datacenter networks. Figure 2.2 shows the distribution of the TCP congestion window (`cwnd`) that we observed while running a latency-sensitive workload [8] on a network with 10G link speeds. We see that the maximum observed `cwnd` size is 32 MSS segments which translates to 46.34 KB. On the other hand, a failure time of 730 μs on a 10G link amounts to 970 KB worth of data transmission. What this means is that, during the time that ShareBackup performs the route recovery, it is possible to lose an entire `cwnd` worth of packets which would cause a TCP retransmission timeout (RTO) and thereby increase the FCT significantly. SQR is complementary to existing route recovery schemes as it helps them to avoid packet loss *during* their route recovery and link failure detection times.

Packet Loss Recovery. Traditionally, packet loss recovery is left to end-point transport. However, for short latency-sensitive flows, end-host recovery incurs FCT

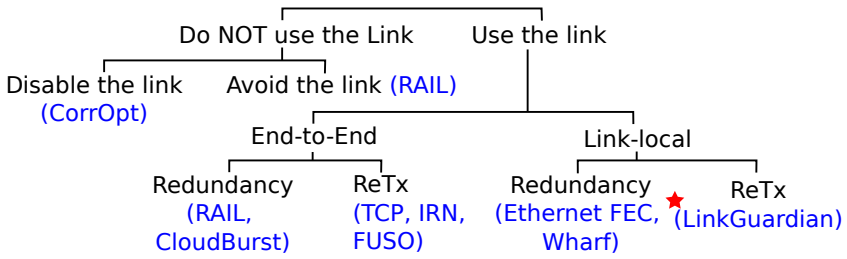


Figure 2.3: Design space for mitigating corruption packet loss due to gray link failures in datacenter networks.

penalty due to packet loss and timeout before recovering the lost packets (c.f. §4.2). Alternatively, end-to-end redundancy approaches can be used [37, 176], where the sender sends duplicate un-ACKed packets on separate paths. However, duplicating packets on the entire path increases the required network bandwidth. Since datacenter networks are often oversubscribed [164], this approach may increase network congestion. Instead of taking up network bandwidth, SQR opportunistically utilizes free packet buffer on the switch to store the duplicate packets. In addition, the end-to-end redundancy methods require changes to the end-host TCP stack. To the best of our knowledge, SQR is the first attempt at in-network packet loss recovery and requires no changes to the end hosts.

Overall, all existing route recovery and packet loss recovery schemes cannot seamlessly divert traffic from a failed path to an alternative path. The main reason is that they do not take into account the inevitable delay and the corresponding packet loss arising from link failure detection and route reconfiguration. Furthermore, since majority of the flows in datacenter networks are small [23], competing approaches of reducing route failure time or flowlet-level switching to alternative paths are not able to mitigate the impact of link failures on short flows. This is precisely the gap that SQR addresses.

2.2.2 Handling Gray Link Failures

In this section, we review the literature related to handling gray link failures. In Figure 2.3, we lay out the design space for mitigating corruption packet loss due to gray link

failures. Below we elaborate on each of the different approaches shown in Figure 2.3.

Disabling the faulty links. The most straightforward and common strategy to deal with corrupting links is not to use them [182, 192]. While this eliminates corruption packet loss, it also reduces network capacity. A recent study of Microsoft datacenters by Zhuo et al. showed that under realistic capacity constraints, some 15% of the corrupting links *cannot be disabled* [192]. So they proposed to find a subset of corrupting links to be disabled such that the impact of the *remaining corrupting links* can be minimized. The strategy of disabling corrupting links has two limitations: first, since not all corrupting links can be disabled, the remaining corrupting links would continue to cause packet drops thereby affecting performance SLOs. Second, disabling a link causes disruption to the rest of the network through packet re-ordering and overall lower network performance [182, 192] during the time it takes for routing and load-balancing protocols to move traffic away from the disabled link and stabilize the rest of the network. CorrOpt [192] is the state-of-the-art solution that employs the strategy of disabling the faulty links. Later in Chapter 5, we propose and evaluate a deployment strategy of our solution LinkGuardian together with CorrOpt. Therefore, we describe CorrOpt in more details below.

CorrOpt. The basic idea of CorrOpt is to disable any gray failure (corrupting) links subject to the capacity constraints of the network. The capacity constraint is specified as the minimum number of valley free paths from a top-of-rack (ToR) switch to the highest level (spine) of the network [192]. For example, in Figure 2.4, we show one "pod"[3] of Facebook's state-of-the-art datacenter network [12]. In Figure 2.4, we can see that each ToR switch has about 192 [(4 fabric switches) $\times$ (48 uplinks)] paths to the spine layer. A capacity constraint of 75% would then mean that every ToR across all pods in the network must have at least 144 paths to the spine layer at all times. Now suppose one of the uplinks to the spine layer for fabric switch 1 (link A in Figure 2.4) starts corrupting

[3]A pod is a unit or building block of modern datacenter networks.

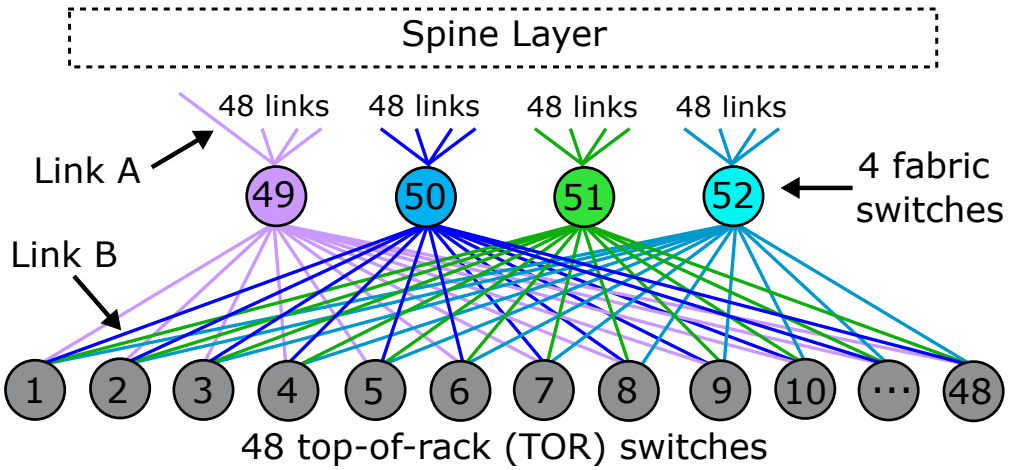


Figure 2.4: A single pod from Facebook's state-of-the-art datacenter network. Image adapted from: Alexey Andreyev, Facebook [12].

packets. In this case, CorrOpt will run its *fast checker* algorithm which checks whether any ToR switch's capacity constraint would be violated if link A is disabled. In this case, all ToR switches in the pod will lose only 1 path and will still have 191 paths remaining to the spine layer. Therefore, CorrOpt will disable this link and schedule it for repair. In the meantime that this link is repaired, suppose the link between ToR switch 1 and fabric switch 2 (link B) starts corrupting packets. Again, CorrOpt will run its fast checker to see if capacity constraints are violated for any ToR switch. In this case, disabling link B will make ToR switch 1 lose another 48 paths to the spine layer as it would be totally disconnected from fabric switch 2. ToR switch 1 would then remain with only 143 [191 - 48] paths to the spine which violates the capacity constraint. Therefore, CorrOpt will not disable link B due to capacity constraint violation and the link will continue to corrupt packets. After a few days, when link A is repaired and enabled, CorrOpt will run its *optimizer* algorithm which checks if any of the remaining corrupting links can be disabled. The goal of the optimizer algorithm is to find a subset of the remaining corrupting links that can now be disabled such it leads to the maximum reduction in the network-wide corruption packet loss. In this example, the *optimizer* algorithm will

find that since link A is enabled, link B can be now disabled after which ToR switch 1 will have 144 paths to the spine layer and thus it will not be violating the capacity constraint. Overall, we see that while CorrOpt is effective in disabling corrupting links, due to the topology structure and the spatial proximity of the corrupting link, there are often times when CorrOpt fails to disable the corrupting links. During such times, the corruption packet loss continues to affect application performance thereby affecting the tail FCTs.

Avoiding the faulty links. Another approach is to avoid corrupting links through source routing or by using virtual network topologies. RAIL allows latency-sensitive applications to avoid lossy links by routing using virtual network topologies [193]. However, in addition to reducing capacity indirectly, it is cumbersome to maintain virtual network topologies as each new corrupting link could require the forwarding entries on several hundred switches across the network to be updated. This approach also requires end-host modifications as applications are required to bind to the appropriate virtual interface.

End-to-end recovery (redundancy). When using a corrupting link is inevitable due to a lack of alternative paths or capacity constraints, end-to-end loss recovery becomes necessary. This can be achieved through proactive redundancy via using end-to-end forward error correction (FEC) [186, 193] or packet duplication [176]. However, this approach adds redundant bytes for *all* the packets across the *entire* path and risks worsening congestion in the network. FEC encoding and decoding also add latency. Further, the required decoding at the receiving end makes it off-limits for supporting one-sided RDMA operations where no CPU is involved on one end.

End-to-end recovery (retransmission). Another option is to simply ignore corrupting links and leave the recovery to the transport endpoints. However, for short latency-sensitive flows on high speed networks, the recovery latency of the end-host transport stack (e.g. TCP) can be much larger than their no-loss flow completion times.

It is possible that latency-sensitive flows can recover faster by using NIC-offloaded [13, 132, 163] and/or multipath [38] transport stacks. Further, IRN proposes to use an adaptive RTO to reduce the recovery delay in case of tail packet loss [132]. However, the fundamental limitation of any end-to-end recovery is that it cannot completely eliminate the use of retransmission timeouts and therefore the recovery delay remains lower-bounded by 1 RTT.

Link-local recovery (redundancy). While proactive link-local redundancy approaches like forward error correction (FEC) are similar in spirit to LinkGuardian, they have several limitations. The Ethernet standards for 25G/100G [90, 91] and 50G/200G/400G [92, 93] specify optional and compulsory FEC at the PHY layer respectively. However, the redundancy parameters are fixed in the standards and cannot be adjusted according to the current loss rate. Wharf [72] also uses link-local FEC but at the level of an Ethernet frame (L2). The main drawback of Wharf is that the redundancy is added to *all* the packets even if the corruption loss rates are very small (see Figure 5.2). When the effective link capacity of the corrupting link is reduced due to FEC overhead, Wharf performs meter-based packet dropping to signal reduced link speed. This will not work well with modern delay-based transport such as TIMELY [131] or Swift [115] and certainly not with loss-sensitive RDMA. Wharf requires FPGA support on switches, and it is unclear if the expensive frame-level FEC encoding/decoding would scale to link speeds of 100G or more.

Link-local recovery (retransmission). Link-level retransmissions is an old idea for wireless networks [1, 2, 19, 88, 89, 148]. SQR [152] is an algorithm that implements link-local retransmission in the datacenter network context, but it is designed to recover packet loss during fail-stop link failures and does not work for corrupting links. LinkGuardian represents a different and unexplored point in the solution design space. Our prior workshop paper [99] investigated the potential of this general idea by implementing a naive out-of-order retransmission mechanism for 10G links. We found that out-of-order

retransmission can completely mask corruption packet loss only on a 10G link. Because TCP has a "reordering tolerance" of 3 packets [11, 24]), if the in-network retransmission can be completed within 3 packet transmissions, then `cwnd` reduction and end-to-end recovery can be avoided. Unfortunately, for higher link speeds such as 100G, the retransmission delay is larger than 2 μs (see Figure 5.10b) while 3 MTU-sized TCP packets can be transmitted faster, i.e., within $\sim$370 ns. This means that to fully mask the packet loss and prevent the performance penalty, we need to preserve packet ordering. Furthermore, our prior work was a work-in-progress and it did not describe a complete solution that: (i) completely masks the corruption packet loss with in-order retransmission (and is hence amenable to RDMA); (ii) handles tail packet loss; (iii) handles consecutive packet loss; (iv) works at high link speeds; and (v) can be deployed effectively on a large-scale network. Therefore, to the best of our knowledge, LinkGuardian is the first complete solution for mitigating corruption packet loss in datacenter networks using link-local retransmission.

Chapter 3

BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks

As described in Section 1.1, congestion events are one of the two main causes for increase in tail FCTs. While significant work has been done to address systematic congestion events, microbursts – which are transient congestion events – can still occur due to lack of tuning and/or complex unanticipated interactions between the flows. Therefore, as argued in section 1.1.1, what is required but is missing from the prior work is a way to continuously and efficiently monitor any microbursts happening in the network.

We address this gap in the literature through the design and implementation of BurstRadar. In this chapter, we first revisit the motivation for monitoring microbursts in the general context of datacenter networking (Section 3.1) before diving into the details of the system design (Section 3.2) and then presenting the evaluation results (Section 3.3). Finally, we conclude the chapter (Section 3.4).

3.1 Introduction

Over the last decade, the performance of datacenter networks has improved significantly [164]. However, service-level guarantees in terms of flow completion times (FCTs) still continues to be a challenging problem as datacenter bandwidths increase and applications become more sophisticated [172]. To achieve more 9's in service-level guarantees, we need to ensure that the tail FCTs remain small and bounded even under unpredictable congestion events. As discussed in §1.1.1, the key to mitigate unpredictable congestion events is to first achieve greater visibility into the network. Modern datacenter networks operate at high-speeds (>10 Gbps) and have ultra-low end-to-end latency (∼10's of μs) [60]. As a result, even small amounts of unpredictable queuing, called *microbursts*, that occur for short periods of time can have a significant impact on application performance and thereby revenue [8].

Microbursts are events of intermittent congestion that last for 10's or 100's of μs. They increase latency and cause network jitter and packet loss in datacenter networks [160]. Common causes include TCP Incast scenarios [8], bursty UDP traffic from an offending flow, as well as TCP segment offloading or application-level batching [107]. The performance degradation arising from microbursts is becoming more common today because link speeds are moving beyond 10 Gbps while switch buffers remain shallow. Traditionally, the impact of microbursts has been greatest for high frequency trading (HFT) applications with reported profit differentials of $100 million per year due to latency advantage of just 1 ms [125]. However, today with low end-to-end latency in datacenters and high SLA requirements by applications, the impact of microbursts is no longer limited to such niche applications. Popular webservices like LinkedIn are reported to have experienced high application latency due to microbursts [103]. Consider an example scenario similar to one reported at LinkedIn [103], where the roll out of a new application service starts to cause microbursts. To address this problem, we first need

to be able to accurately detect the occurrence of these microbursts and identify the culprit service/application who added a large number of packets in quick succession to the microburst's queue build up. Once the culprit service is identified, the burstiness of the flows from that service could be fixed by introducing packet pacing or by other means. Notice that once the cause is determined, in most cases, the fix for microbursts is usually not very difficult. However, the key challenge lies in detecting the occurrence of these microbursts and identifying the culprit flows. Therefore, in this thesis, we focus on the detection of microbursts and collection of the required telemetry information that would allow finding the root cause(s).

The extremely low timescales make it impossible for traditional sampling-based techniques such as Netflow [47] and sFlow [150] to even detect the occurrence of microbursts. Some existing commercial solutions [45, 102, 138] are able to detect microbursts, but provide no information about the cause. Recent advances in programmable dataplanes [27] and dataplane telemetry have led to proposals for In-Band Telemetry (INT) [77, 111] that embed telemetry information into each packet and enable debugging for several network issues including microbursts. However, since microbursts are unpredictable [188], it is wasteful to use INT to monitor them as it would require the telemetry information for every single packet in the network to be captured and processed, while only a small number of packets contribute to microbursts. Basat et al. [22] show that INT's approach of embedding telemetry information into the packets can lead to 25% increase in average FCTs since the increase in packet size worsens congestion queue build-ups.

In this thesis, we demonstrate that programmable dataplanes can be used to detect microbursts more efficiently by capturing the telemetry information of only the packets involved in microbursts. Our system, called *BurstRadar*, builds on the out-of-band approach for exporting telemetry information [82]. Our key insight is that microbursts are localized to a port's egress queue. This makes all the information required for detecting and characterizing a microburst available *together* on a single switch. Unlike

an INT-based approach [77], by detecting a microburst directly on the switch where it happens, we can avoid the computations and delays arising from having to correlate monitoring information from different points in the network. BurstRadar's key idea is to take a "snapshot" of all the packets in a queue at every time instant when the queue grows larger than an operator-specified threshold and export this snapshot information out of the switch dataplane for root cause analysis. The "snapshot" contains the header information of all the packets at that instant in the queue, their arrival and departure times from the queue, as well as the queue size when the packets were enqueued or dequeued from the queue. BurstRadar uses a *Snapshot algorithm* (§3.2.1) that runs on a per-packet basis in the egress pipeline of the switch dataplane to capture such queue snapshots i.e. precisely identify the packets belonging to the queue snapshots. It then uses egress packet cloning (§3.2.2) to generate on-demand *courier* packets for transporting this information together.

While our approach is relatively straightforward given existing programmable dataplane architectures, we made three observations from our design and implementation. First, we need a strategy to temporarily store telemetry information before it can be transferred to the courier packets. Second, there is a sizable delay in the generation of courier packets and it depends on packet size, among other factors. Third, it is possible that if there are multiple simultaneous microbursts on different egress ports, telemetry information for some packets involved might be lost. BurstRadar provisions the temporal storage by implementing a ring buffer using the transactional stateful memory available in the dataplane (§3.2.3). We then handle the issues of courier packet delays and multiple simultaneous microbursts by sizing the ring buffer appropriately (§3.2.3). Overall, the complete BurstRadar solution works as follows: The Snapshot algorithm first identifies the packets belonging to queue snapshots and for each such packet, it writes the telemetry information to the ring buffer while also signaling the courier packet generator to generate a courier packet. After a small delay, when the courier packet is generated,

the courier packet reads the telemetry information from the ring buffer and transports it to a cluster of monitoring servers for root cause analysis.

We have implemented BurstRadar on a Barefoot Tofino [139] switch and evaluated it on a multi-gigabit hardware testbed using utilization burst distributions from Facebook's production network [188]. Our results show that even with microbursts occurring as frequently as every 200 μs, BurstRadar processes 10 times less telemetry information compared to INT [77, 111], while providing all information to fully characterize microbursts and identify contributing flows. BurstRadar captures telemetry information for all packets contributing to microbursts, even with bursts occurring simultaneously on multiple egress ports. Further, it achieves real-time detection of microbursts at multi-gigabit link speeds.

3.2 System Design

Our key idea is to first detect a microburst in the dataplane and then capture a *snapshot* of telemetry information of all the involved packets. This information allows queue composition analysis to identify the culprit flow(s), and burst profiling to know burst characteristics such as duration, queue build-up/drain rates, etc. Detecting a microburst is relatively easy with queuing telemetry information provided by modern programmable switching ASICs [139]. However, taking a *snapshot* of all the packets involved in the microburst and further exporting this information in an out-of-band manner is non-trivial for three reasons. First, the switching ASIC's "Buffer and Queuing Engine" (BQE) does not provide any support to peek into the contents of any queue, and so the *snapshot* needs to be captured from *outside* the BQE. Second, any logic in the programmable pipelines outside the BQE can only execute on a per-packet basis. Third, exporting the snapshot information requires on-demand generation of new *courier* packets in the dataplane and transferring of snapshot information to these *courier* packets. These

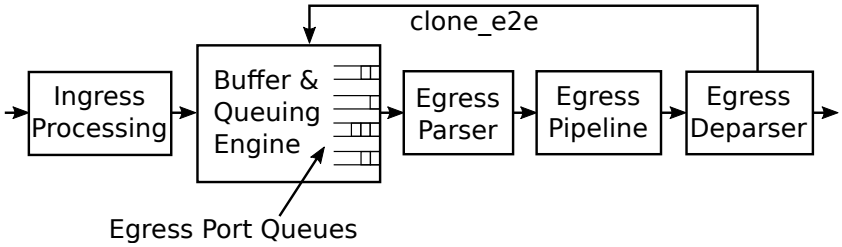


Figure 3.1: General architecture of a programmable switching ASIC [49]

challenges exist for today's pipelined-architecture switches which are commonly used in datacenter networks for their low-latency performance [27, 117, 139].

Figure 3.1 shows the general architecture of a programmable switching ASIC. BurstRadar runs in the egress pipeline of each switch in the network. It consists of three functional components: (i) Snapshot Algorithm, (ii) Courier Packet Generation, and (iii) Ring Buffer. The Snapshot algorithm first determines the packets that are involved in a queuing microburst and *marks* them. The marking is done via a metadata header and does not modify the original packet. For each marked packet, a courier packet is generated to transport the marked packet's telemetry information via the switch's mirror port. The ring buffer provides temporary storage to facilitate the transfer of telemetry information from the *marked* packets to the courier packets. The telemetry information for each marked packet consists of the packet 5-tuple, the ingress and egress timestamps, and the queue depths at the time of enqueue and dequeue (`enqQdepth` and `deqQdepth`). Courier packets are processed at the monitoring servers connected to the mirror port infrastructure. In the following subsections, we describe these three components in more detail.

3.2.1 Snapshot Algorithm

While BurstRadar can monitor all queuing microburst events, small queuing events that cause negligible increase in application latency are generally not of interest to the operator. Therefore, BurstRadar allows the operator to specify a *latency-increase*

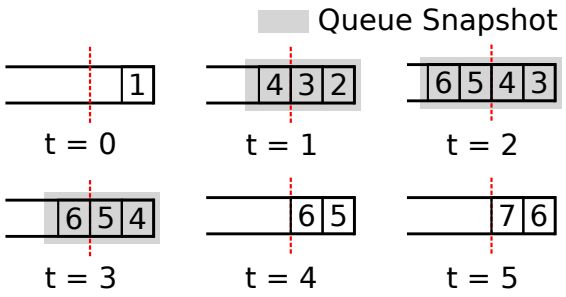


Figure 3.2: Evolution of an example queuing microburst at different instants in time

threshold (specified as a percentage). For example, if the network's no-queuing RTT is 50 μs, then the operator may specify a threshold of 30% which translates to a minimum latency increase of 15 μs. BurstRadar would then ignore any microbursts that incur less than 15 μs of delay. The threshold is set on a per-switch basis and the exact value depends on the maximum delay that can be tolerated by the deployed applications. For latency-sensitive applications like web services, the threshold could be set to a small fraction of the RTT. On the other hand, it could be a few multiples of the RTT for throughput-intensive applications like Hadoop.

We define a *queue snapshot* to be the set of packets present in the queue when the queue-induced delay is above the operator-specified threshold. Figure 3.2 shows the evolution of a toy queuing microburst at different instants in time. At time instant t=1, the queue length exceeds the threshold (dotted line). Thus the snapshot of the queue at this instant consists of packets {2,3,4}. Similarly at t=2, the queue snapshot consists of packets {3,4,5,6}. At t=3, the queue starts to drain, but we still have a queue snapshot consisting of packets {4,5,6}. At t=4 and beyond, the queue length falls below the threshold and we stop taking snapshots. In other words, a single queuing microburst event consists of multiple overlapping queue snapshots. Note that at t=5, an additional packet #7 enters the queue but is not a part of any of the queue snapshots.

Since egress port queues are a part of the BQE (refer Figure 3.1), it would be easy to capture queue snapshots inside the BQE. However, the BQE in today's programmable

Algorithm 1: Queue Snapshot Algorithm

```
   Input: threshold
   Initialization: bytesRemaining = 0;
1  foreach pkt in egressPipeline do
2      if deqQdepth > threshold then
3          bytesRemaining = deqQdepth − size(pkt);
4          mark(pkt);
5      else
6          if bytesRemaining > 0 then
7              bytesRemaining = bytesRemaining − size(pkt);
8              mark(pkt);
   end
```

switching ASICs doesn't provide any such functionality, but provides the queuing telemetry information (`enqQdepth` and `deqQdepth`) for each packet leaving the BQE. The `enqQdepth` refers to the queue size at the time instant when the packet is enqueued, and similarly the `deqQdepth` refers to the queue size when the packet is dequeued. Our Snapshot algorithm uses this telemetry information and runs outside the BQE in the egress pipeline.

Since the egress pipeline follows a per-packet execution model, the Snapshot algorithm (Algorithm 1) needs to decide (*mark*) whether a packet entering the egress pipeline belongs to any queue snapshot or not. For the example microburst in Figure 3.2, we need to mark the packets #2 to #6, but not #7. To decide if a packet should be marked, we consider the queue length when a packet is dequeued (`deqQdepth`). There are two possible cases: (i) The `deqQdepth` is greater than the threshold, or (ii) The `deqQdepth` is less than or equal to the threshold. In the former, it is clear that the packet belongs to at least one of the queue snapshots. For example in Figure 3.2, packet #2's `deqQdepth` is greater than the threshold and thus packet #2 would be *marked* at t=1. Similarly, packets #3 and #4 would be marked at t=3 and t=4 respectively. At each of these time instants, the Snapshot algorithm also maintains and updates the `bytesRemaining` in the queue (line 3). For example, at t=3, the `bytesRemaining` would be set to the

total bytes of packets #5 and #6. In the latter case when the reported `deqQdepth` is less than or equal to the threshold, only the packets equivalent of `bytesRemaining` would be marked (lines 6-8). In the example, packets #5 and #6 would be marked due to `bytesRemaining` set by packet #4; but packet #7 would not be marked. Essentially, when a queue drains below the threshold, `bytesRemaining` helps to track and mark the packets (#5 and #6) that were part of the last queue snapshot (snapshot at t=3).

The telemetry information for these marked packets is then stored in a ring buffer (§3.2.3) and the courier packet generation logic (§3.2.2) is also signaled to generate a courier packet to transport the telemetry information to a cluster of monitoring servers.

3.2.2 Courier Packet Generation

BurstRadar generates a courier packet for each *marked* packet on demand. To do this, BurstRadar uses the *clone egress to egress* or `clone_e2e` primitive provided by programmable switching ASICs [49]. The `clone_e2e` primitive makes a copy of the exiting regular packet and places it in the egress queue of the mirror port (see Figure 3.1). The courier packet is also appropriately truncated to remove the original payload.

3.2.3 Ring Buffer

A ring buffer is designed using the transactional stateful memory available in the egress pipeline and exposed by the P4 programming language [26] as *register* arrays. The ring buffer acts as temporary storage for the telemetry information of marked packets until it can be copied into the courier packets. We use two circular pointers – write pointer and read pointer – for pointing to the next index of the register arrays to write or read. The Snapshot algorithm uses the write pointer to write to the ring buffer while the courier packets uses the read pointer to read from the ring buffer. Since the Snapshot algorithm first writes to the ring buffer before signaling the generation of the corresponding courier packet, the read pointer always trails the write pointer.

Ring Buffer Sizing. We found that the first read from the ring buffer by a courier packet happens only after a *cloning delay*. During a microburst, the interval between the first and second writes to the ring buffer is mainly determined by the serialization delay of the first packet. If the serialization delay of the first packet is smaller than the cloning delay, the second packet in the microburst will write to the ring buffer before the first courier packet performs the first read. Therefore, the ring buffer needs to be large enough to store the information for the marked packets passing through the pipeline before the first read by the courier packets.

Figure 3.3 compares the cloning delay to the serialization delay (at 10 Gbps link speed) for packets of different sizes. For 64 byte packets, the cloning delay (270 ns) is more than five times the serialization delay (51.20 ns). This means that in the worst case of having all 64 byte packets in a microburst, more than five writes would be made to the ring buffer before the first read happens. For our ASIC implementation, we found that factors[1] other than the packet size also affect the cloning delay. Accordingly, we found (by measurement) the required minimum ring buffer sizes for 10 Gbps and 25 Gbps link speeds to be 26 entries and 32 entries, respectively. Since the size of each entry is 29 bytes, these requirements translate to 754 bytes and 928 bytes, which are very small given the SRAM memory sizes (up to 100 MB) in today's ASICs [129].

Concurrent Microbursts. Since the egress pipeline is shared among the ports, it serves the egress port queues in a round-robin manner. Therefore, if multiple egress ports simultaneously experience microbursts, in one scheduling round of the egress pipeline, there would be multiple writes to the ring buffer while only a single read due to a single mirror port. This seems to suggest that a very large ring buffer might be required to handle multiple concurrent microbursts. However, as we show in §3.3.2, due to statistical multiplexing, a ring buffer with 1k entries is sufficient in practice to handle 10 simultaneous microbursts without any overwrites.

[1]Investigation of all factors will be addressed in a separate measurement study.

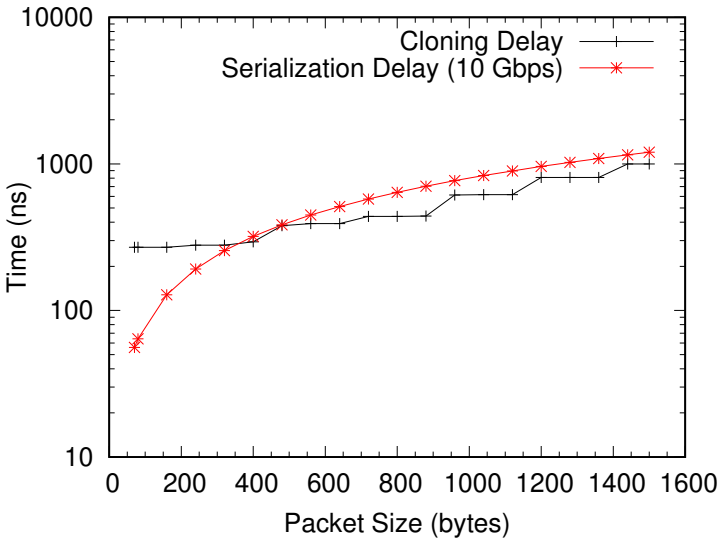


Figure 3.3: Cloning and serialization delay for packets of different sizes

3.2.4 Implementation

BurstRadar can be implemented with small modifications to fixed function switching ASICs or programmed on modern programmable switching ASICs [36, 95, 139]. We implemented BurstRadar on a Barefoot Tofino switch [139] in about 550 lines of P4 [26] code. The operator-specified latency-increase threshold is stored in a *register* in the dataplane and can be dynamically configured by the control plane. The Snapshot algorithm and the ring buffer are implemented using a sequence of exact match-action tables. Arithmetic operations are facilitated by stateful ALUs.

Switching ASICs (fixed or programmable) provision memory for buffered packets using fixed size memory buckets or *segments* [138]. Therefore, the reported `deqQdepth` is expressed in terms of number of segments. Snapshot algorithm converts the `deqQdepth` from segments to bytes to compute `bytesRemaining` (line 3 in Algorithm 1). This conversion results in excess `bytesRemaining` compared to the actual remaining bytes, causing BurstRadar to mark extra packets towards the *tail-end* (lines 6-8 in Algorithm 1) of a microburst. For example, if the segment size is 160 bytes and the queue consists of

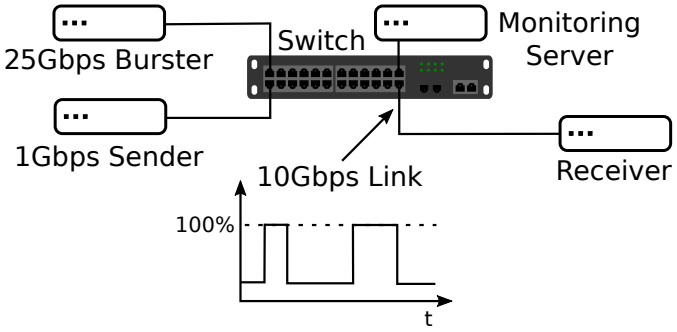


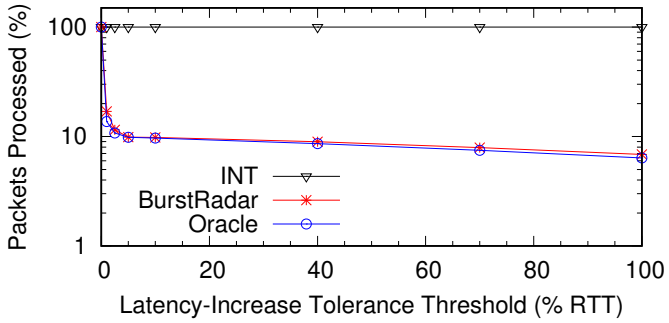
Figure 3.4: Testbed setup

a single 161 byte packet, then the reported `deqQdepth` of 2 segments would be converted to 320 bytes, instead of the actual 161 bytes.

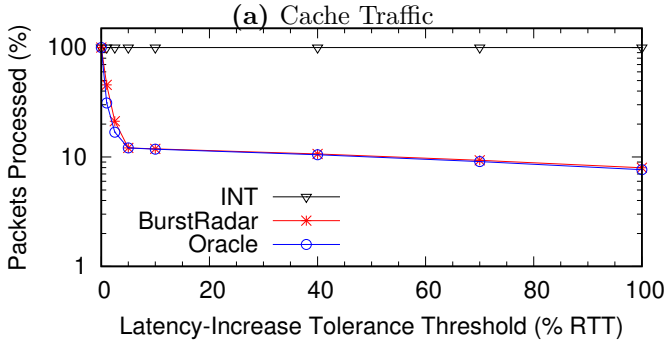
3.3 Evaluation

The evaluation of our BurstRadar prototype is centered around answering three questions. First, how efficient is BurstRadar, given that it selectively *snapshots* microburst queues? Second, how well does BurstRadar handle multiple simultaneous microbursts? And finally, what is the cost of BurstRadar in terms of hardware resources required in the switching ASIC?

Testbed. The evaluation experiments were conducted in our hardware testbed which consists of a Barefoot Tofino [139] switch and commodity servers equipped with Intel XXV710 (25/10 Gbps) and Intel X710 (10 Gbps) NIC cards. To precisely generate network traffic at μs resolution and cause microbursts as per the input network traces, we wrote our own traffic generator application (450 lines of C++) using the PcapPlusPlus library [159] with DPDK [63] as the datapath. The testbed is organized in a topology as shown in Figure 3.4. Based on the data in [188], the sender continuously sends 1 Gbps background traffic keeping the utilization of the test link under 10% for 90% of the time. The burster emulates different sources of microburst, sending bursts at 25 Gbps such that the queuing at the switch's egress buffer (and the subsequent 100% link utilization)



(a) Cache Traffic



(b) Web Traffic

Figure 3.5: Fraction of total number of packets processed for different latency-increase thresholds

follows the distributions for duration and inter-arrival times as per the input trace.

Network Traces. Data on the frequency and duration of *queuing* microbursts is currently not available publicly. Therefore, we took reference from the data on *link utilization* bursts in a Facebook datacenter [188]. It provides the distribution of duration, inter-arrival times and packet size for utilization bursts when the link utilization spikes above 50%. We can safely assume that this is the worst case upper bound on the duration and inter-arrival times for *queuing* microbursts, which entail 100% link utilization on the egress link. We used the traffic data from two latency-sensitive applications – web, and in-memory cache – to generate 10-second long traces.

Methodology. We compare BurstRadar to INT [77, 111] and to an offline Oracle algorithm. The Oracle algorithm has access to the telemetry information of all the packets and is thus able to capture queue snapshots as if they were captured by the BQE (c.f. §3.2.1). It represents the optimal solution.

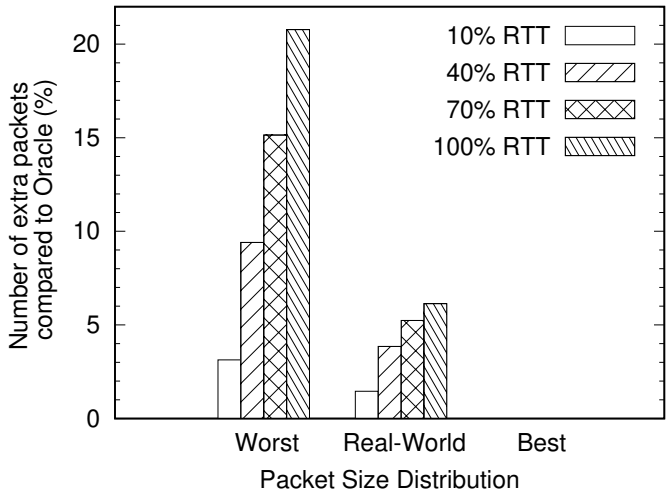


Figure 3.6: Number of extra packets marked compared to the Oracle solution for different packet size distributions (Cache Traffic)

3.3.1 Efficiency

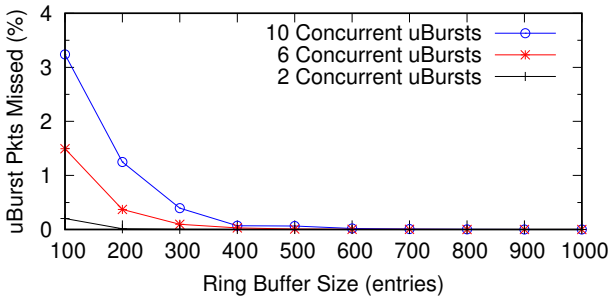
We quantify the overhead of continuous microbursts monitoring in terms of the fraction of total number of packets that are required to be processed by the monitoring system. We compare the overhead incurred by BurstRadar, INT and the Oracle algorithm for the cache and web traffic in Figure 3.5. We observe in Figure 3.5a that even for a low latency-increase threshold of 5% RTT, BurstRadar is 10 times more efficient than INT. Since the RTT is approximately 25 μs in our testbed, this threshold translates to 1.25 μs of queuing delay, or 1562.5 bytes worth of queuing at 10 Gbps. We verified with our experiments that this threshold only filters out packets that are not involved in microbursts. In practice, latency sensitive applications might not require such a low threshold and therefore the overhead for BurstRadar would be even lower. Note that at a latency-increase threshold of 0% RTT, BurstRadar would be equivalent to INT as telemetry information for every single packet is processed. The efficiency result for web traffic is similar to cache traffic as shown in Figure 3.5b.

Overhead of Extra Packets. While BurstRadar is more efficient than INT in terms of the number of packets processed, it does process a few extra packets due to the segments to bytes conversion of `deqQdepth` (see §3.2.4). The number of extra packets identified by BurstRadar depends on the packet size distribution and the segment size of the ASIC's packet buffer. The worst case occurs when every packet is exactly one byte larger than the segment size, thereby causing each packet to occupy two segments. The best case occurs when the size of all packets is an integer multiple of the segment size.

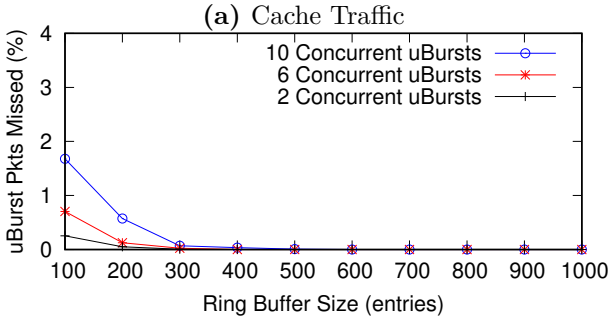
In Figure 3.6, we plot the number of extra packets identified by BurstRadar compared to the Oracle algorithm for the cache workload with different packet size distributions: worst, real-world and best. For real-world, we use the cache workload's original packet size distribution [188]. We note that while the worst case shows about 21% extra packets compared to the Oracle, the number of extra packets is typically only about 6% as shown by the real-world case. In the best case, there are no extra packets. Figure 3.6 also shows that a larger latency-increase threshold leads to a higher proportion of extra packets. This is because for a given packet-size distribution, larger the latency-increase threshold, larger the number of packets in the *remaining* queue below the threshold, with each packet contributing extra bytes to `bytesRemaining`. The real-world overhead for web traffic is lower than the cache traffic due to larger packet sizes [188] and is omitted because of space constraints.

3.3.2 Handling Concurrent Microbursts

As discussed in §3.2.3, multiple concurrent microbursts can result in a higher rate of writes to the ring buffer than the rate of reads. If the ring buffer size is not sufficiently large, ring buffer overwrites may occur, leading to the loss of telemetry information for some of the *marked* packets. Currently, no data is available on how often we should expect concurrent microbursts at different egress ports for a switch. Therefore, we



(a) Cache Traffic



(b) Web Traffic

Figure 3.7: Fraction of microburst packets missed with concurrent microbursts for different ring buffer size

simulate[2] microburst traffic on multiple ports of a switch using 10-second long traces of cache and web traffic from [188]. In Figure 3.7, we plot the fraction of microburst packets missed by BurstRadar for different ring buffer sizes (10 to 1k entries) when microbursts occur on 2, 6 and 10 ports concurrently. With just 300 entries, BurstRadar is expected to be able to handle 10 simultaneous microbursts (cache traffic) with a packet miss rate lower than 1%. About 1000 entries are required to reduce the miss rate to absolute 0%. This suggests that BurstRadar is resilient and can handle simultaneous microbursts with a modestly-sized ring buffer.

[2]This experiment is currently not supported by our testbed due to lack of equipment to generate multiple concurrent microburst traffic.

Table 3.1: Hardware resource consumption of BurstRadar (ring buffer size of 1k entries) compared to the baseline switch.p4

Resource	switch.p4	BurstRadar	Combined
Match Crossbar	50.13%	3.39%	53.52%
Hash Bits	32.35%	4.83%	37.18%
SRAM	29.79%	4.06%	33.85%
TCAM	28.47%	0.69%	29.16%
VLIW Actions	34.64%	4.69%	39.33%
Stateful ALUs	15.63%	12.5%	28.13%

3.3.3 Resource Utilization

In Table 3.1, we compare the hardware resources required by our BurstRadar prototype (with a ring buffer of 1k entries) to that required by a production (closed-source) version of switch.p4. The switch.p4 is a baseline P4 program that implements various networking features (L2/L3 forwarding, VLAN, QoS, ACL, etc.) for a typical datacenter ToR switch. A simplified open-source version of switch.p4 is available at [48]. We note that BurstRadar's overall resource consumption is low for various hardware resources. BurstRadar consumes a relatively larger proportion (12.5%) of stateful ALUs as they are used for the computations in our Snapshot algorithm and for managing the ring buffer pointers. The SRAM is used for the exact match-action tables and for implementing the ring buffer. Despite the ring buffer size of 1k entries, BurstRadar's SRAM requirements remain low. Also, the combined usage of all resources by switch.p4 and BurstRadar is well below 100%. This means that BurstRadar can easily fit on top of switch.p4.

3.4 Summary

Detecting microbursts in a datacenter network and identifying the contributing flows is difficult because microbursts are unpredictable and last for 10's or 100's of μs. BurstRadar leverages programmable switching ASICs to implement continuous and efficient monitoring of microbursts by capturing the telemetry information of only the packets involved

in microbursts. Our testbed evaluation using production network traces demonstrates that BurstRadar can detect microbursts with 10 times less overhead compared to existing approaches and is resilient to simultaneous microbursts. This chapter describes our BurstRadar prototype, as well as the design decisions and considerations in dataplane packet cloning and ring buffer sizing. Our work demonstrates that modern programmable ASICs have made it practical to detect and characterize microbursts at multi-gigabit link speeds in high-speed datacenter networks.

BurstRadar primarily captures local queue information which is sufficient to find the root cause of microbursts in most cases. However, in cases involving transient root causes, finding the root cause may require global information. For example, data from BurstRadar can help to conclude that packets from certain set of servers arrived at the same time and that caused the microburst. However, it cannot immediately tell "why" they arrived at the same time – were they sent in a synchronous manner by the servers or they got synchronized within the network. Answering this question requires capturing global telemetry information and our follow-up work called SyNDB [105] (outside of this thesis) is able to achieve this.

Chapter 4

SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks

In the previous chapter, we described BurstRadar that contributes towards mitigating the increase in tail FCTs due to unexpected congestion events. As described in Section 1.1 and shown in Figure 1.2, in order to ensure bounded tail FCTs, we also need to handle link failure events. Now link failure events are of two types: (i) fail-stop: where the entire link goes "down" and no packets can travel in either direction; and (ii) gray: where the link is "up" but randomly drops a few packets due to packet corruption. Both types of link failures cause packet loss that lead to increased tail FCTs due retransmission delays and retransmission timeouts. In this chapter, we address the problem of increased tail FCTs due to fail-stop link failures through the design and implementation of **S**hared **Q**ueue **R**ing (**SQR**). We first describe in more details the problem of fail-stop link failures in datacenter networks (Section 4.1) followed by a measurement study (Section 4.2) which demonstrates that existing link failure management techniques fall short

of keeping the tail FCTs low under link failures. We then present the detailed system design of SQR (Section 4.3) before presenting the evaluation results (Section 4.4). Finally, we discuss other related issues (Section 4.5) before concluding the chapter (Section 4.6). For the remaining of this chapter, we refer to fail-stop link failure simply as "link failure".

4.1 Introduction

Datacenter computing is dominated by user-facing services such as web search, e-commerce, recommendation systems and advertising [8]. These are *soft real-time* applications because they are latency-sensitive and the failure to meet the response deadline can adversely impact user experience and thus revenue [8]. Such application-level deadlines can be translated into flow completion time (FCT) targets for the network communication between the *worker* processes that work together to serve the user requests [180]. There have been many proposals to reduce the FCTs of latency-sensitive flows for user-facing soft real-time applications under normal network conditions [8, 10, 87, 131, 136, 180]. In this chapter, we study the problem of reducing FCTs in the presence of link failures.

Datacenter networks typically use commodity hardware components mainly to save costs. These include line cards, transceivers, cables, connectors, etc. Failure of any of these components leads to failure of the corresponding network links. Now, the failure rate of these individual components is very small – about 10^{-5} failures per hour [67]. However, in large warehouse-scale datacenters, since there are hundreds of thousands of these components, the network-wide failure rate can be very large (10^{1}). As a result, link failures are common in datacenter networks. A large-scale study across a Microsoft datacenter reported 136 link failures per day at the 95^{th} percentile [73]. The resulting packet loss from link failures has an outsized impact on short latency-sensitive flows. Such flows typically operate with a small TCP congestion window so when there is packet loss, the TCP receivers cannot send enough duplicate ACKs within one RTT [41]. As a

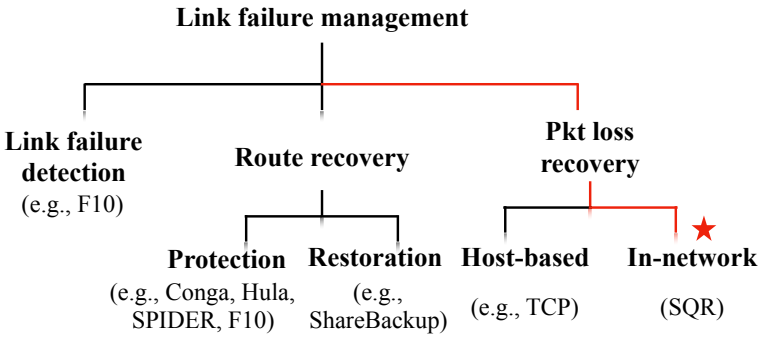


Figure 4.1: Design space for link failure management.

result, fast retransmission is rarely triggered and the lost packets are often recovered via retransmission timeouts (RTOs) [184]. Such timeout events result in much larger delays than the lifespan of the short flows and significantly increase FCT [185].

A large number of approaches have previously been proposed to reduce the impact of link failures, including fast re-routing [35, 147, 158, 162, 174], flowlet-based load balancing [7, 109] and re-configurable topologies [122, 181]. All these approaches inevitably rely on *link failure detection* which has a minimum delay. Among them, the state-of-the-art ShareBackup [181] takes as little as 730 μs to recover from a link failure and it relies on F10's link failure detection technique [122] which has a delay of about 30 μs. This total delay of 760 μs on a 10 Gbps link translates to about 950 KB, and some 600 1500-byte packets could be lost. Even if we can reroute immediately after detecting the link failure (30 μs) using a pre-computed backup path, some 25 1500-byte packets could still potentially be lost. This suggests that while existing proposals can achieve low FCTs under normal network conditions, they cannot maintain or keep these low FCTs *stable* under link failures, even when using state-of-the-art link failure recovery techniques. In other words, in the face of link failures, the datacenter network stack today is unable to provide any tight bounds or strong reliability guarantees (up to five or six 9's) on FCTs or the network latency.

We observe that to completely mask the effect of packet loss and the resulting long

recovery latency, the network has to be responsible for packet loss recovery, instead of relying on end-to-end recovery. To this end, we propose **S**hared **Q**ueue **R**ing (SQR), an on-switch mechanism to recover packets that could be lost during the period from the detection of a link failure to the completion of the subsequent network reconfiguration. The overall workflow for SQR is as follows: SQR makes copies of recently transmitted packets and caches them for a configurable amount of time. In the meantime, a link failure detection mechanism continuously looks for link failures. If no link failure occurs, SQR simply drops the cached copies of packets. However, if a link failure is detected, the failure detection mechanism informs SQR about the same and also activates a route reconfiguration mechanism. In this case, SQR continuous to buffer the packet copies until the route reconfiguration mechanism signals SQR that a backup path has been established. SQR then retransmits the cached packet copies on the backup path. SQR is therefore complimentary to existing methods of link failure detection and route reconfiguration as shown in Figure 4.1. Furthermore, it operates locally and independently at each switch without requiring any coordination with other switches.

It is not possible to know in advance if a link will fail when a packet is sent, since link failures occur randomly and cannot be predicted [73]. We define the *route failure time* to be the worst-case time taken to detect a link failure and for the network to recover. We observe that by estimating the upper bound on the route failure time, a switch can cache a copy of recently sent packets for this duration. Then, in the event of a link failure, we can avoid packet loss by retransmitting the cached copy of these previously transmitted packets on the backup path. Naively, this can be implemented as a *delayed queue* that temporarily delays (stores) every packet passing through it for a configurable amount of time. When the link fails, we can retransmit the cached packets from this delayed queue. An ideal delayed queue is where each packet in the queue has a future timestamp at which it would be sent and the queue is ordered by these timestamps. Unfortunately, no queuing engine today readily provides the queuing discipline required to realize such a

Table 4.1: ASIC packet buffer trends

ASIC	Year	Packet Buffer
Trident+ [28]	2010	9 MB
Trident II [29]	2013	12 MB
Trident II+ [56]	2015	16 MB
Tomahawk+ [31]	2016	22 MB
Tomahawk II [32]	2017	42 MB

delayed queue. Furthermore, existing queuing engines, including those in programmable ASICs, cannot be programmed to implement a custom packet scheduling algorithm that implements a delayed queue. To realize a delayed queue in any other way, the basic primitive required is packet storage. In a switch dataplane, even a programmable one, the packets can only be stored in the packet buffer of the queuing engine [27]. This packet buffer storage can only be utilized by placing packets into the default FIFO queues, which send out packets as fast as possible without introducing any delay.

In this chapter, we describe a technique to emulate a delayed queue in the dataplane of a programmable switch. Our emulated delayed queue differs slightly from the ideal delayed queue, wherein it delays the packet for "at least" the specified amount of time, instead of "exactly" the specified amount of time. We do so by retaining a copy of a sent packet in a FIFO queue. If this packet reaches the head of the queue before being sufficiently delayed, we use egress processing to route this packet back into the FIFO queue. While this approach is, in principle, sufficient to emulate a delayed queue, it is challenging to ensure that no packet is missed out and the packets are retransmitted *in order*. Furthermore, there are two costs involved – the egress pipeline processing required to build and maintain the emulated delayed queue, and the additional packet buffer required for the packets in the delayed queue (i.e. the cached packets). A naive implementation could inflict additional egress processing delays on other flows going through the switch. SQR avoids this with a *Multi-Queue Ring* architecture that exploits unused egress processing capacity. The egress pipeline is provisioned to support all

ports at full packet rate. In practice, most networks will almost always have spare egress processing capacity available [188]. We only use the idle ports so that other traffic passing through the switch is uninterrupted.

We implemented SQR on a Barefoot Tofino [139] switch[1]. We show using experiments on a hardware testbed using trace-driven workloads that:

- SQR can completely mask the effect of link failures from end-point transport by preventing packet loss;
- Coupled with current link failure detection (F10 [122]) and route reconfiguration schemes (ShareBackup [181]), SQR can reduce the tail FCT by 10 to 1000 times for web and data mining workloads in the presence of link failures; and
- SQR's overhead in terms of packet buffer consumption, additional egress processing and ASIC hardware resources is low, thereby demonstrating the feasibility of our solution.

Gill et al. observed that network redundancy is not entirely effective in reducing the impact of link failures [73]. Our work addresses this gap by enabling a *seamless hand-off* of packets from a failed route to an alternative route, thereby fully exploiting available multi-path redundancy. To the best of our knowledge, we show, for the very first time, that it is possible to handle link failures without a single packet being lost or reordered in a multi-gigabit datacenter network[2]. Our proposed approach was not previously feasible because switches would not have enough packet buffer to cache packets for the route failure time. However, recent innovations have substantially reduced the route failure time (65 ms in Portland [140] to 760 μs in ShareBackup [181]) so that the number of packets to be cached is significantly reduced. On the other hand, on-chip shared packet buffer for switching ASICs has increased more than fourfold over the last 5-7 years (see

[1] A simple version of SQR is available at: `https://github.com/NUS-SNL/sqr`

[2] As long as SQR uses the correct estimates for the worst-case link failure detection and route recovery delays.

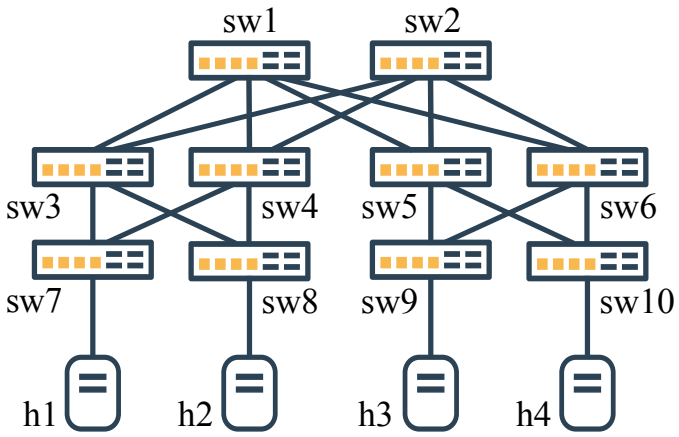


Figure 4.2: Testbed.

Table 4.1), making in-network seamless packet hand-off practical.

4.2 Motivation

Link failures are dominated by connection problems such as cabling and carrier signaling/timing issues [62]. Gill et al. observed that link failures were more common than device failures, and some 136 link failures were observed daily at the 95^{th} percentile [73]. Link failures usually last for a few minutes and exhibit a high variability in their rate of occurrence.

For any solution that tries to minimize the effects of link failures, there are two main delays involved: (i) *link failure detection delay*, the time it takes to detect that a link has failed, and (ii) *network reconfiguration delay*, the time required to reconfigure the network and restore route connectivity in response to the link failure. Together we refer to the sum of these delays as the *route failure time*. In this section, we show that although the route failure times have reduced from 65 ms in Portland [140] to 760 μs in ShareBackup [181], short latency-sensitive flows still suffer from high FCTs when there are link failures. To the best of our knowledge, ShareBackup currently has the lowest reported route reconfiguration time.

Setup. We do not have access to an optical switch, and so we *emulated* ShareBackup's behavior in our testbed by disabling a link and enabling it again after ShareBackup's route reconfiguration time. We refer to this simulation of ShareBackup as ShareBackup′ or SB′. Our testbed (Figure 4.2) consists of a fat-tree topology built using a partitioned Barefoot Tofino switch (similar to [106]) and Intel Xeon servers equipped with Intel X710 NICs. All links are 10 Gbps and the network RTT between the hosts is about 100 μs. Each host runs Linux kernel 4.13.0 with TCP CUBIC. SACK is enabled and RTO_{min} is set to the smallest possible value of 4 ms. Host $h2$ sends short, latency-sensitive ($\leq$100 KBytes [7]) TCP flows to host $h4$ via the path $sw8 \rightarrow sw4 \rightarrow sw2 \rightarrow sw6 \rightarrow sw10 \rightarrow h4$. The flow sizes are drawn from the distribution of a web search workload [8]. The flows are sent one at a time with no other network traffic. Since the FCT of a small flow is less than 2 ms in normal case, we inject link failures between switches $sw6$ and $sw10$ every 20 ms[3] to ensure that each flow experiences link failure at most once. To emulate ShareBackup's route reconfiguration, we use precise dataplane timer mechanisms to generate a link failure that lasts for exactly 760 μs. We use a *deflect-on-drop* switch dataplane mechanism to identify the flows affected by link failures.

FCTs under Link Failures. Figure 4.3 shows the FCTs of the flows where failure-affected flows form three distinct horizontal clusters[4]. The cluster of FCT values around 1 second is due to the SYN or SYN-ACK packet loss since the default retransmission timeout (RTO) for these packets is set to 1 second [156]. The middle two clusters of FCT values ($\sim$10 ms and $\sim$100 ms) are due to RTOs being triggered either due to *tail losses* in a `cwnd` or the complete loss of all packets in a `cwnd`. For failure affected flows, we did not observe any fast retransmissions. Overall, we see that when there are contiguous packet losses due to link failures, even with state-of-the-art fast recovery mechanisms like ShareBackup, the FCTs for short flows can increase by several orders of magnitude.

[3]This failure injection rate only facilitates faster experimentation and does not influence the results.
[4]The small gap in the flow size is an artifact of the web search workload [8] that we use.

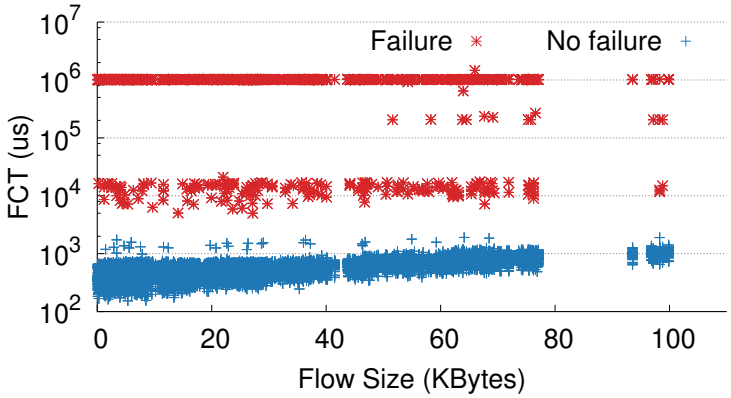


Figure 4.3: FCTs of latency-sensitive web search flows [8] under link failures with ShareBackup as route recovery mechanism.

To further understand this result, we measured the TCP `cwnd` sizes for the above flows under no link failure (no loss) conditions. We found that, at 90^{th} percentile, the `cwnd` size is about 10 MSS segments which is the default initial `cwnd` size on Linux [44]. The maximum observed `cwnd` size was 32 MSS segments which translates to 46,336 bytes with MSS being 1448 bytes. However, at 10 Gbps link speed, a route failure time of 760 μs translates to 950,000 bytes, i.e. 656 MSS segments after accounting for the Ethernet preamble, framing, and inter-frame gap. Therefore, the route is in the failed state for a much larger duration than the time it would take for a `cwnd` worth of packets to traverse a link in the network. This implies that it is very unlikely to have packet losses as "holes" *within* a `cwnd` so as to trigger fast transmissions. In our experiment with short flows, link failures always triggered expensive RTOs resulting in significantly longer FCTs.

The results presented above also hold true for other deployed TCP variants (DCTCP [8], TIMELY [131]) since they all employ the same mechanism for handling packet loss. In summary, our results (which concur with the results in [37]) show that the tail and whole window losses dominate in case of short flows, triggering RTOs and inflating FCTs under link failures. Therefore, to reduce tail FCTs under link failures, we need to avoid RTOs.

Discussion. The impact of RTOs can be alleviated to an extent by using microsecond-

level RTO_{min} [101], which requires significant modifications to the end-host network stack [175]. A small RTO_{min} however risks reducing throughput due to spurious retransmissions [151] and leads to increased overall packet loss for incast-like scenarios [101]. Deciding on the right value for RTO_{min} is tricky and it is typically set at 5 ms in production datacenters [37, 40]. At this value, the majority of latency-sensitive flows are small enough to complete in one RTT [75] and therefore under link failures they would take at least twice as long to complete, irrespective of the value of RTO_{min}.

4.3 SQR Design

In §4.2, we argued that to eliminate high FCTs under link failures, we need to avoid RTOs. SQR therefore focuses on fast in-network recovery of packets lost during the route failure time, without involving the end hosts. Our key idea is to continuously *cache* a small number of recently transmitted packets on a switch and in the event of a link failure, retransmit them on the appropriate *backup* network path.

SQR runs entirely in the dataplane of an individual switch. Our design assumes the Portable Switch Architecture (PSA) [49] consisting of an ingress pipeline, a Buffer and Queuing Engine (BQE), and an egress pipeline. When an incoming packet enters the ingress pipeline, the primary egress port is determined by the network's routing scheme. Subsequently, the packets from the latency-sensitive applications will be marked if it belongs to a latency-sensitive flow[5] that needs to be protected by SQR. The packet passes through the BQE normally and when it arrives at the egress pipeline, it is subjected to SQR's processing if it is marked.

In the egress pipeline, SQR by default forwards a packet to the destination port and be proceeded normally. However, if a packet is *marked*, SQR performs the additional task of creating a copy of the packet and caches the copy for a time duration equal to the

[5]Latency-sensitive applications can request SQR's protection by using a pre-defined set of TCP port numbers, IP Header TOS bits or VLAN IDs.

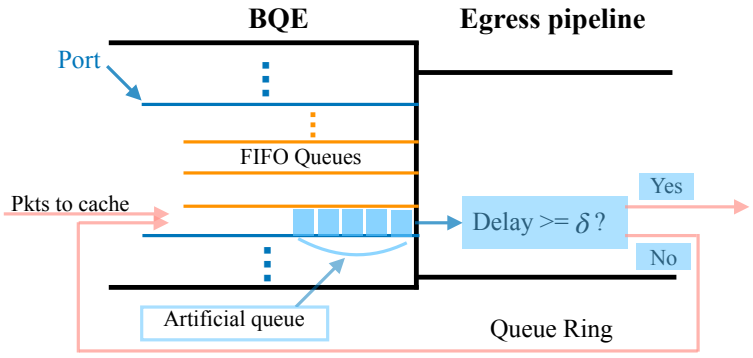


Figure 4.4: Caching packets on switch using a FIFO queue.

link failure detection delay (§4.2). By doing so, SQR ensures that packets are not lost if a link fails later. After this delay, SQR checks if the cached packet's primary egress port (on which the original packet was sent) is still operational. If the link is up, then it means that the original packet was transmitted successfully and the cached copy of the packet is dropped. However, if the link is down, then the original packet was likely lost and a copy of the packet is cached again for time equal to the *network configuration delay* (§4.2). This additional delay allows the network to configure the backup path without losing the cached packets. After this second delay, the cached packets are sent on the backup path (port).

SQR's operation requires the following information in the switch dataplane: (i) The link status of the ports (*up* or *down*), (ii) the *backup* port (route) for each *primary* port (route) to a destination top-of-the-rack switch. SQR integrates with a link failure detection mechanism such as the one used in F10 [122] to update and maintain the status of the ports (albeit after a delay). It also integrates with a route recovery scheme (e.g. ShareBackup [181]) to determine the *backup* port for each *primary* port.

4.3.1 Caching Packets on the Switch.

Conceptually, the caching of packets can be achieved with a *delayed queue*, where each individual packet entering the queue is delayed for a fixed *minimum* amount of time (termed as *delay time*) before it leaves the queue. Unfortunately, there is no such primitive in the current switching ASICs. Furthermore, the queuing engines, including those in programmable ASICs do not support programming such custom scheduling to realize a delayed queue inside the queuing engine. In addition, packet storage, which is required to realize a delayed queue, is only available inside the queuing engine in the form of FIFO queues. Therefore, it is not straightforward to realize a delayed queue in existing switching ASICs.

SQR achieves the delayed queue functionality using a "Queue Ring" that combines the BQE's FIFO queues, egress pipeline processing and high-resolution timestamping. Today's programmable switches support multiple FIFO queues per port (all inside the BQE) and the BQE handles all queues across all ports on the switch. The high-level idea (shown in Figure 4.4) is to place the packets to be cached inside a spare FIFO queue of a port on the switch (queue selection details in §4.3.2). When the FIFO queue transmits the cached packet at a later time, high-resolution timestamping is used to check if the packet has been delayed for the required duration δ. If the packet is not sufficiently delayed ($delay < \delta$), the egress pipeline sends the cached packet back to the FIFO queue. Once a packet is sufficiently delayed ($delay >= \delta$) after passing through the FIFO queue one or more times, it exits the Queue Ring. This helps to build up an *artificial* queue of cached packets, since *effectively* no packet exits without being sufficiently delayed. In the steady state, where new packets enter the Queue Ring at a fixed rate R, the artificial queue build-up remains fixed and equal to $R \times \delta$. Notice that each packet accumulates delays from two sources – (i) the queuing delay due to the artificial queue build-up, and (ii) the egress processing delay incurred in sending a packet

back to the FIFO queue. Hereafter we will refer to the FIFO queue used to implement a Queue Ring as the *caching queue*.

4.3.2 Multi-Queue Ring Architecture

Our Queue Ring approach utilizes the egress processing of the port associated with the caching queue to emulate the delayed queue behavior. Unfortunately, the processing of these cached packets may affect the normal traffic passing through other queues of that port. Therefore, to minimize the impact on existing traffic, we do not use the same port for packet caching. Instead, SQR assigns one queue from the multiple queues [46] of each port as a *caching queue* and spreads the queued packets across a set of these caching queues. In particular, when a cached packet is to be sent back to a FIFO queue for additional delay, SQR dynamically chooses the caching queue that belongs to a port with the least utilization. We refer to this architecture that consists of multiple caching queues from the BQE that are connected to each other by the egress pipeline to form a *ring* as the Multi-Queue Ring (see Figure 4.5). We exploit the fact that while the egress pipeline is provisioned to support all ports at full packet rate, there is almost always spare egress processing capacity available in the switch under typical network load conditions [188]. The spare capacity, however, is available on different ports at different times. Using a ring of multiple queues allows SQR to exploit the spare capacity by dynamically changing the set of low utilization ports.

An artifact of SQR's Multi-Queue Ring architecture is that when the cached packets exit after being buffered, they do not exit in the same order as they originally entered the Multi-Queue Ring. Therefore, in the event of a link failure, the exiting cached packets need to be ordered before they are sent to the backup port. To do so, SQR uses a counter-based packet sequencing mechanism. SQR's Multi-Queue Ring architecture is implemented with three components running in the egress pipeline (also shown in Figure 4.5): (i) a delay timer (§4.3.3) to keep track of each cached packet's elapsed time,

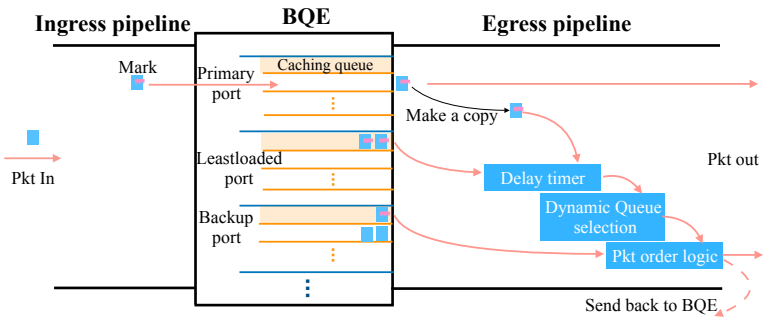


Figure 4.5: Multi-Queue Ring architecture.

(ii) a queue selection algorithm (§4.3.4) to dynamically choose the next caching queue, and (iii) a packet order logic (§4.3.5) to order the cached packets before re-transmission.

4.3.3 Delay Timer

The delay timer first computes how long each packet has been buffered in the Multi-Queue Ring (called `ElapsedTime`). To do so, when a copy of the original packet is created, the delay timer attaches the egress timestamp provided by the dataplane (called `StartEgressTstamp`) to the copied (cached) packet as metadata. As the cached packet passes through the Multi-Queue Ring, it enters the egress pipeline one or more times. Each time in the egress pipeline, the delay timer calculates the time elapsed so far (`ElapsedTime`) by taking the difference between the current egress timestamp (`CurrentEgressTstamp`) and the packet's `StartEgressTstamp`. The delay timer then compares the `ElapsedTime` with the required delay time (δ) to check if the packet has been buffered for at least the delay time. If so, the delay timer would set the `DelayEnough` field in the packet (later used by the queue selection algorithm in §4.3.4). The delay timer logic is summarized in Algorithm 2. Because of limited bit-width (n bits) clock register in the switch dataplane, the calculation needs to handle cases with value wrap around.

Delay Time (δ). This is the time for which each copied (cached) packet needs to

Algorithm 2: Delay Timer.

```
   Initialization: ElapsedTime = 0, pkt.DelayEnough = 0;
1  foreach marked pkt in egress pipeline do
2  |  diff = CurrentEgressTstamp − StartEgressTstamp;
3  |  if diff > 0 then
4  |  |  ElapsedTime = diff;
5  |  else
6  |  |  ElapsedTime = 2^n + diff;
7  |  if ElapsedTime => δ then
8  |  |  pkt.DelayEnough = 1;
   end
```

be buffered on the switch. Since there is a delay in detecting link failures, δ is initially set equal to the upper bound of the link failure detection delay. When a link failure is detected, SQR dynamically increases δ by value equal to the network reconfiguration delay so as to hold the cached packets until the network reconfiguration is complete. Since the total packets being buffered on the switch is proportional to δ (c.f. §4.3), its value determines SQR's packet buffer requirement (§4.4.4).

4.3.4 Dynamic Queue Selection

Recall from §4.3.2 that SQR designates one queue on each port as the caching queue. In the Multi-Queue Ring, each time a cached packet is to be sent from the egress pipeline back to the BQE, the queue selection logic (Algorithm 3) decides to which caching queue to forward the packet. As the goal is to minimize the impact on other traffic, SQR selects the next caching queue from a port which has the least utilization *at the current moment* (called the `LeastLoadedPort`). A packet is sent to the `LeastLoadedPort` in the following cases: (i) if it is a freshly made copy of an original packet *and* the `PrimaryPort` is UP, or (ii) if it is an already cached packet that has not been sufficiently delayed. A sufficiently delayed cached packet (as indicated by the Delay Timer in §4.3.3) is dropped if the primary link is up. If the primary link is down and the incoming packet is an original

Algorithm 3: Dynamic Queue Selection.

```
   Input: PrimaryPort, LeastLoadedPort, BackupPort
 1 foreach marked pkt in egress pipeline do
 2 |   if PrimaryPort == UP then
 3 |   |   if cached pkt then
 4 |   |   |   if pkt.DelayEnough ! = 1 then
 5 |   |   |   |   Send pkt to the LeastLoadedPort;
 6 |   |   |   else
 7 |   |   |   |   Drop cached pkt;
 8 |   |   else
 9 |   |   |   Make a copy and send the copy to LeastLoadedPort;
10 |   else
11 |   |   if cached pkt then
12 |   |   |   if pkt.DelayEnough ! = 1 then
13 |   |   |   |   Send pkt to the LeastLoadedPort;
14 |   |   |   else
15 |   |   |   |   Send pkt to BackupPort;
16 |   |   else
17 |   |   |   Send pkt to BackupPort;
   end
```

packet or a sufficiently delayed cached packet, it is sent to the caching queue of the backup port for retransmission.

Tracking Port Utilization. SQR tracks the egress utilization of all the ports by maintaining a moving window of the number of bytes transmitted on each port. The size of the window is the time interval over which the number of transmitted bytes are accumulated. We discuss window sizing in §4.3.6. SQR maintains a `LeastLoadedPort` and the corresponding `LeastUtilization`. When an original packet arrives at the egress pipeline, the utilization of its egress port is updated. If this utilization is lower than the `LeastUtilization`, SQR will update the `LeastUtilization` to the current utilization and the `LeastLoadedPort` to the current egress port. When an original packet is transmitted on the `LeastLoadedPort`, SQR will also update the value of `LeastUtilization`.

Algorithm 4: Packet Order Logic.

```
   Input: NextPktTag, PrimaryPort, BackupPort
 1 foreach marked pkt in egress pipeline do
 2 |   if PrimaryPort == UP then
 3 |   |   if pkt.DelayEnough == 1 then
 4 |   |   |   NextPktTag = PktTag + 1;
 5 |   else
 6 |   |   if PktTag == NextPktTag then
 7 |   |   |   NextPktTag + = 1;
 8 |   |   else
 9 |   |   |   if PktTag > NextPktTag then
10 |   |   |   |   Send pkt to BackupPort;
   end
```

4.3.5 Packet Order Logic

When a link failure happens, the delay timer (§4.3.3) and the dynamic queue selection (§4.3.4) would send the cached copies of recently transmitted packets to the backup port (path). However, since cached packets are circulated through a *ring* of queues, the order in which they are sent to the backup port may not be the same as the original arrival sequence. To ensure that packet ordering is preserved, the packet order logic (Algorithm 4) first needs to know the original ordering of the packets. To achieve this, the packet order logic consists of a monotonically increasing packet counter in the egress pipeline. When an original packet to be protected by SQR enters the egress pipeline, the counter value (`PktTag`) is added to the packet as metadata and gets copied to the corresponding cached packet. The packet order logic also maintains an expected next counter number (`NextPktTag`). Both the `PktTag` and the `NextPktTag` are used to ensure correct packet ordering as following: (i) if the cached packet's `PktTag` is equal to the expected `NextPktTag`, SQR just sends the packet and updates the `NextPktTag` (lines 6-7); (ii) if the cached packet's `PktTag` is larger than the `NextPktTag`, it will send this packet back to the backup port's caching queue and wait for the packet with the correct `PktTag` (line 9-10) to be sent first. When a cached packet with a `PktTag` leaves the

switch due to either being dropped after sufficient buffering or sent on the backup path, SQR updates the `NextPktTag` (lines 4, 7). Since the cached packets are ordered before being sent, this may add extra delay on recovery time (§4.4.2).

4.3.6 Implementation

We implemented SQR on a Barefoot Tofino switch [139] in about 1,100 lines of P4 code. A common action performed by SQR is to send a packet from the egress pipeline back to the BQE. This action is achieved with two primitives, egress-to-egress cloning (also called mirroring) and packet drop. For each cached packet, the SQR metadata is added when the cached packet is first created and is removed before the packet is sent out of the switch. The SQR metadata contains three fields: (i) `PktTag`: used by packet order logic for reordering (§4.3.5); (ii) `StartEgressTstamp`: used by delay timer to record when the cached packet was created (§4.3.3); (iii) `PrimaryPort`: used by queue selection logic to track the cached packet's primary port (§4.3.4).

The delay timer, queue selection logic and the packet order logic are implemented using a series of exact match-action tables and stateful ALUs. The *delay time* is stored in a dataplane register and can be dynamically configured based on the link failure detection mechanism being used. For computing link utilization (§4.3.4), we set the moving window size larger than the network RTT to avoid sensitivity to transient sub-RTT traffic bursts [7]. At the same time, we also avoid setting the window so large that it would aggregate the bytes of entire short flows and make SQR sluggish to react to the flow churn. Since the network RTT in our testbed is about 100 μs and the minimum FCT in our evaluation workloads is about 157 μs, we used a window size of 150 μs in our prototype. The `LeastLoadedPort` and `LeastUtilization` are also maintained using dataplane registers. We note that SQR's implementation requires standard primitives such as egress mirroring, encap/decap (for SQR metadata), registers and match-action tables which are specified in the PSA [49] and also available in fixed-function ASICs.

Therefore, SQR can be implemented on any programmable ASIC based on the PSA [49] or it could be baked into fixed-function ASICs.

4.4 Performance Evaluation

We evaluate our SQR prototype by answering three questions: (1) How effective is SQR in masking link failures from end-point TCP stack, such that RTOs will not be triggered? (2) When SQR is integrated with other network reconfiguration systems (e.g. ShareBackup), how much is the reduction in FCTs under link failures for latency-sensitive workloads? (3) What is the cost (overhead) of SQR in terms of effect on other traffic and consumption of resources in the switch hardware? We perform the evaluation on the same hardware testbed as described in §4.2 unless otherwise mentioned.

4.4.1 Experimental setup

Workloads. We consider two empirical workloads with short flows taken from production datacenters: a web search workload [8] and a data mining workload [75]. The CDF of flow sizes for these two workloads is shown in Figure 4.6. For both the workloads, we consider flow sizes up to 100 KB since these represent latency-sensitive flows [7]. We use a server-client model in which a server sends TCP flows of sizes drawn from these two distributions to a client. Specifically, in our testbed (Figure 4.2), host $h2$ sends TCP flows to host $h4$.

Background Traffic. We run the Spark TPC-H decision support benchmark to generate background traffic. It contains a suite of database queries running against a 12 GB database on each worker. The master node is $h4$ (see Figure 4.2) which communicates with the slave nodes $h1$ and $h2$ via the paths $sw10 \rightarrow sw6 \rightarrow sw2 \rightarrow sw4 \rightarrow sw7$ and $sw10 \rightarrow sw6 \rightarrow sw2 \rightarrow sw4 \rightarrow sw8$, respectively. The query job is submitted to the master node and multiple tasks run on the three nodes.

Baseline Schemes. Recall that SQR integrates with a link failure detection and a network reconfiguration scheme (§4.3). We consider the link failure detection method suggested in F10 [122] (detection delay = 30 μs) and two different network reconfiguration methods: ShareBackup [181] (SB′) and local rerouting (LRR), in the following configurations:

1. **SB′:** As explained in §4.2, SB′ is our emulated version of ShareBackup that takes an additional 730 μs to restore network connectivity via backup switches after a link failure is detected.
2. **LRR: L**ocal **R**e**R**outing runs a path probing protocol [7, 109] to proactively-determine a backup port for each primary port. When the link on a primary port is detected to be down, the traffic is immediately re-routed to the backup port thus incurring no network reconfiguration delay.
3. **SB′+SQR:** SQR integrated with SB′ involves setting the backup port to be the primary port itself since ShareBackup uses optical switching to restore connectivity on the same port. The initial delay time is 30 μs and is increased to 760 μs on link failure detection (§4.3.3).
4. **LRR+SQR:** SQR integrated with LRR involves setting the backup ports to the ones determined proactively. The delay time is 30 μs at all times.

Link Failure Model. SQR helps with link failures where multiple paths are available. Therefore, we inject a link failure every 20 ms[6] between $sw6$ and $sw10$ while $h2$ is sending traffic to $h4$ (Figure 4.2). Similar to §4.2, for SB′ we restore the failed link after the route failure time (760 μs).

[6]This failure injection rate only facilitates faster experimentation and does not influence the results.

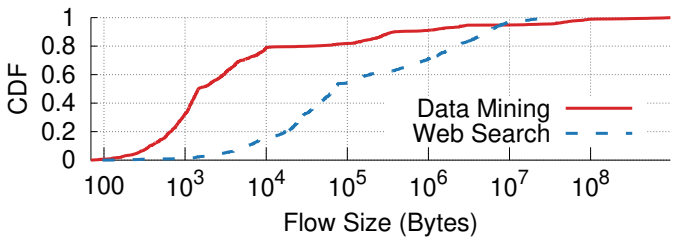
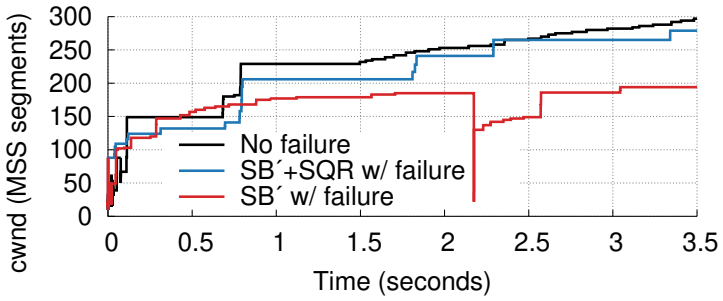


Figure 4.6: Flow size distributions used in evaluation.

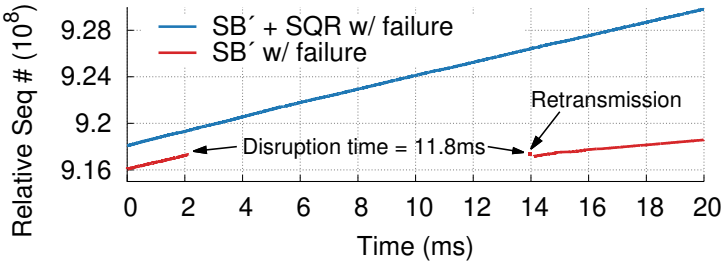
4.4.2 Masking Link Failures from TCP

First, we evaluate SQR's effectiveness in masking link failures from the end-point transport protocol (TCP). We compare TCP's behavior under link failure when running SB′ alone to that when running SB′ along with SQR. $h2$ starts an iperf client to send TCP traffic to an iperf server running on $h4$ (see Figure 4.2). To properly observe the TCP sequence numbers from captured traces, we set TSO off (only for this experiment). We use `n2disk` [143] to capture the packet traces and the `tcp_probe` kernel module to capture the TCP sender's connection statistics. About 2 seconds after starting the flow, we inject a link failure on the link between the switches $sw6$ and $sw10$. Figure 4.7 shows one instance of the result. Results are similar when link failure is introduced at a different location in the network.

Figure 4.7a shows the evolution of the TCP sender's `cwnd`. We see that with SB′ alone, the TCP sender reduces its `cwnd` size drastically when there is a packet loss due to link failure. However, when SB′ is enhanced with SQR, the link failure has no impact on the TCP sender and the `cwnd` grows like the no-failure case. In Figure 4.7b, we plot the TCP stream's sequence number of packets as sent by the sender. With SB′ alone, when the link fails, the TCP sender stops sending due to absence of ACKs and times out leading to a disruption time of about 12 ms. By the time the TCP sender recovers from the timeout, SB′ has already restored the connectivity and the sender resumes by first retransmitting the lost packets. However, when SB′ is coupled with SQR, the TCP



(a) TCP congestion window



(b) TCP sequence number (zoomed view after 2 seconds)

Figure 4.7: TCP sender's cwnd and seq number progression for SB′ with and without SQR. Link failure occurs after about 2 seconds.

sender is not affected by the link failure and the TCP sequence number grows smoothly.

Recovery Time. While Figure 4.7 shows the TCP sender's perspective, the perspective from a TCP receiver is different. Upon link failure, while the route is being reconfigured, SQR *holds* the packet transmission thereby introducing a time small gap. This small time gap, called the *recovery time*, is an unavoidable effect seen by a TCP receiver. Figure 4.8 shows the CDF of the recovery time for over 30,000 TCP flows where the link failure is masked in a SB′+SQR configuration. The recovery time is larger than SB′ route failure time (760 μs) in about 90% instances for two reasons. First, SQR needs to reorder the packets before retransmission which adds some additional recovery delay. Second, the underlying delayed queue causes each individual packet to be delayed for a time *at least* equal to the *worst-case* route failure time. The packets that are delayed for longer than the *actual* route failure time are those that are not lost and would be delivered to the receiver again. Retransmitting these *extra packets* also contributes to the

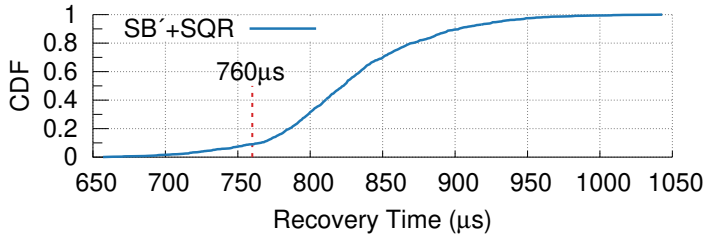


Figure 4.8: Recovery time.

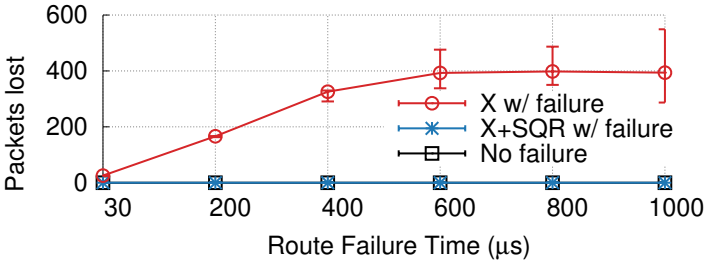


Figure 4.9: Number of packets lost for different route failure time.

additional recovery time. Note that these extra packets do not affect the TCP receiver's state and the resultant FCT for short flows [25]. In about 10% instances, the recovery time was lower than 760 μs. We believe that in these instances, due to "natural" gaps in the packet transmission, the packets arrived after a link failed and before the route was successfully recovered, thereby getting buffered for less than 760 μs.

Packet Loss. The number of packets lost during a link failure depends on the recovery scheme's route failure time. A scheme with a higher route failure time would stress SQR. Figure 4.9 shows the number of packets lost for a generic route recovery scheme `X`, whose route failure time varies from 30 μs (LRR) to 1000 μs (F10 [122]). Beyond the route failure time of 600 μs, the number of lost packets does not increase as TCP loses almost the whole `cwnd` and the transmission is stalled. When `X` is coupled with SQR, the packet loss remains zero even when the route failure time increases.

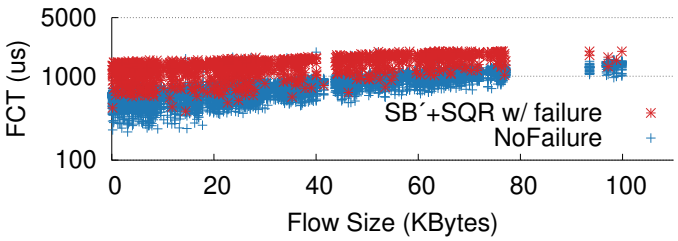


Figure 4.10: FCTs of latency-sensitive web search flows [8] under link failures with SB′+SQR as route recovery mechanism.

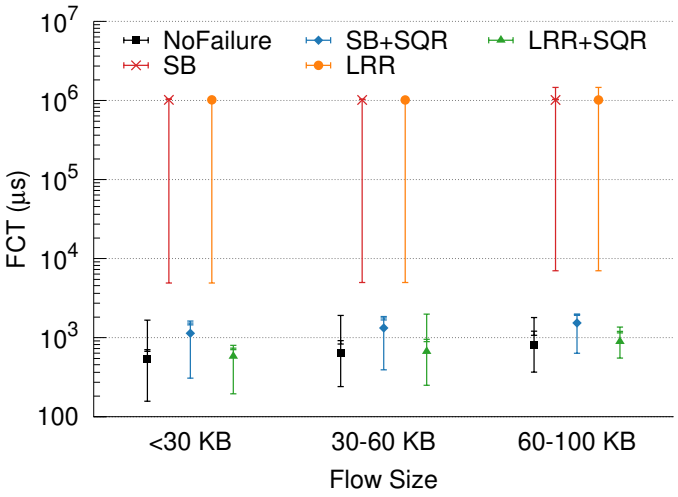
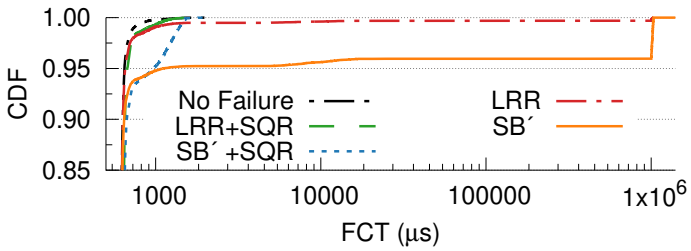


Figure 4.11: FCTs of failure-hit web search flows [8] compared to no failure.

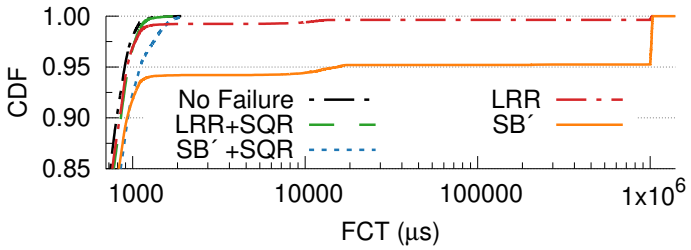
4.4.3 Latency-sensitive Workloads

Next, we evaluate how effective is SQR at keeping FCTs low for latency-sensitive workloads under link failures. We use 1,000 different flow sizes from the web and data mining workloads (§4.4.1) and send 30 flows of each flow size yielding a total of 30,000 flows. The flows are sent from $h2$ to $h4$ while the link between $sw6$ and $sw10$ is brought down every 20 ms (see Figure 4.2). The total route failure time is 30 μs for LRR and 760 μs for SB′.

We first focus on the FCTs of flows which faced link failures i.e. we ignore the flows that were not affected by a link failure. We showed in §4.2 that even with SB′, the FCTs



(a) Data mining workload



(b) Web search workload

Figure 4.12: CDF of FCTs for two workloads under link failures.

can increase by several orders of magnitude when there are link failures (see Figure 4.3). Figure 4.10 shows that when SB′ is coupled with SQR, the FCTs for the failure-hit flows are only slightly higher than the FCTs of no failure flows. Figure 4.11 shows the FCTs for failure-hit web search flows when running SB′ and LRR schemes with and without SQR. We show the results for three different ranges of flow sizes. The vertical bars show the minimum, median, 95$^{\text{th}}$ percentile, 99$^{\text{th}}$ percentile and the maximum values of FCT. We observe that when coupled with SQR, the tail FCTs of failure-hit flows for both SB′ and LRR are reduced by about 3 to 4 orders of magnitude. If the packets of a flow arrive after the link has failed and before the route is reconfigured, the *recovery time* (see §4.4.2) of these packets will be less than the route failure time. Therefore, even though SB′ has a 760 μs route failure time, the minimum and median values of FCT for SB′+SQR are only about 200 μs higher than the no failure or LRR+SQR scenarios.

Figure 4.12 shows the FCT distribution for all the 30,000 flows involved in an experiment run, including those not affected by link failures. For both the data mining and

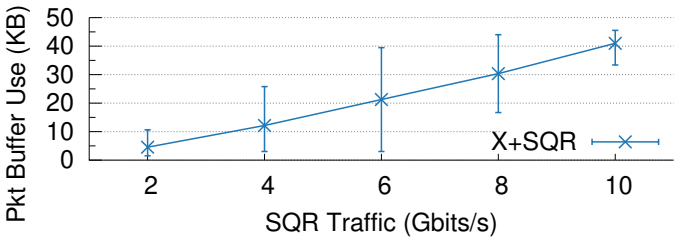


Figure 4.13: Steady-state packet buffer consumption (per-port).

web search workloads, the tail FCT of SB′ is slightly worse than LRR. This is because SB′ has a longer route failure time compared to LRR. While SQR helps in cutting down the overall tail FCT for both SB′ and LRR, its reduction in FCT for LRR is slightly more than that for SB′. This is because although SQR prevents packet loss, it inflicts a *recovery time* delay (see §4.4.2) which is higher for SB′ than that for LRR.

4.4.4 Overhead

Finally, we investigate the overheads incurred by SQR by measuring: (i) the packet buffer consumption, (ii) the reduction in switch throughput; (iii) the additional hop latency on the switch; and (iv) the hardware resources required when implemented on a programmable switch.

Packet Buffer Consumption. SQR uses the switch packet buffer to cache packets for the *delay time* (see §4.3.3). Since SQR uses a ring of queues, the packet buffer consumption at any time is equal to the total number of cached packets across the different caching queues. To measure the packet buffer consumption, we configured SQR's Multi-Queue Ring to use only a single queue (just for measurement). Then using the *queue depth* provided by the programmable dataplane, we measured the depth of the queue to obtain the packet buffer consumption. During steady-state (no link failure), SQR only caches packets for the link failure detection delay. Therefore, its steady-state packet buffer consumption depends only on the link failure detection mechanism. For a generic route recovery scheme (which we denote with `X`), Figure 4.13 shows how the

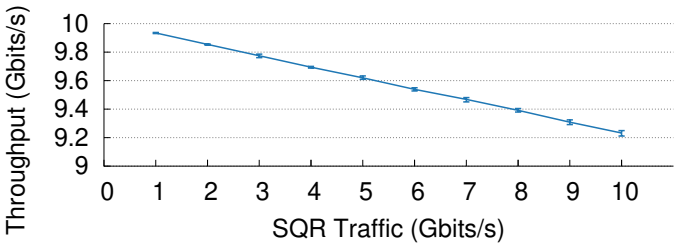


Figure 4.14: Impact of SQR processing on normal line-rate traffic.

steady-state packet buffer consumption (per-port) increases with SQR traffic volume while using F10's link failure detection mechanism (detection delay = 30 μs). Clearly, the packet buffer consumption increases with increase in SQR traffic. For a 10 Gbps link, SQR will only need to handle up to 10 Gbps traffic in the worst case, even when there is an *incast* (>10 Gbps) of incoming latency-sensitive traffic. This is because SQR protects traffic on the egress link whose rate is constrained by the link speed. Therefore, the worst case packet buffer consumption per SQR-enabled port is given by,

$$\text{Worst Case Pkt Buffer} = \text{Link Speed} \times \text{Delay Time} \tag{4.1}$$

From equation 4.1, we would expect the worst-case steady-state buffer consumption for a 10 Gbps port with a 30 μs failure detection delay to be 37.5 KB. This matches our experimental results in Figure 4.13. However, when a link failure is detected, the delay time is increased to 760 μs in case of SB′+SQR. In this instance, according to equation 4.1, the buffer consumption for SB′+SQR would be 950 KB in the worst case. Fortunately, the failed-state is very short-lived (and will last only until the route is reconfigured), after which SQR returns to steady-state caching.

Impact of SQR on Normal Traffic. SQR incurs some additional egress pipeline processing to send insufficiently delayed cached packets back to the BQE (§4.3.1). To measure the impact of SQR's processing (maintaining a delayed queue) on the normal traffic, we configure SQR's Multi-Queue Ring to contain only a single caching queue on a port, say $\mathtt{p}_1$. We then start line rate TCP (10 Gbps) background traffic whose egress

port on the switch is also $\texttt{p}_1$. The background traffic uses a queue on port $\texttt{p}_1$ that is different from the SQR's caching queue, but has the same scheduling priority. A SQR-enabled flow (SQR traffic) is then started on another port $\texttt{p}_2$. All packets from this flow are cached using $\texttt{p}_1$'s caching queue.

Figure 4.14 shows the throughput of the line-rate background traffic for different rates of SQR traffic. We see that even at 10 Gbps, SQR traffic will occupy only about 750 Mbps of egress processing. This means that as long as the normal traffic is less than 9.25 Gbps, it will not be impacted by the processing overhead of 10 Gbps SQR traffic. In other words, a single 10 Gbps port can simultaneously support 9.25 Gbps of normal traffic and egress processing of 10 Gbps SQR traffic. Given that SQR uses dynamic queue selection (§4.3.4) to utilize only the `LeastLoadedPort` each time the next caching queue in the Multi-Queue Ring is chosen, the likelihood of negatively impacting the normal traffic is very low.

Switch Processing Latency. SQR is mostly non-intrusive to the SQR-protected original traffic, but incurs some additional dataplane processing. To measure the latency added by this additional processing, we send traffic from $h1$ to $h2$ along $sw7 \rightarrow sw4 \rightarrow sw8$ (Figure 4.2). When a packet arrives at the ingress pipeline of $sw7$ or $sw4$, we add the ingress timestamp (*IngressTs*) to it. The difference between the two *IngressTs* of adjacent switches is the *hop latency*. We found that, on average SQR adds a negligible 4.3 ns of *additional* hop latency compared to a P4 program that does only L3 forwarding.

Hardware Resources Requirements. In Table 4.2, we compare the hardware resources required by SQR to that required by *switch.p4*, which is a close-source production P4 program that implements all the network features of a typical datacenter ToR switch. SQR uses a relatively larger proportion of stateful ALUs for operations such as calculating the `ElapsedTime`, determining the `LeastLoadedPort`, and comparing the `PktTag` with the `NextPktTag`. SQR's logic is achieved using exact match-action tables which require SRAM. However, SQR's overall resource consumption remains low.

Table 4.2: Resource consumption of SQR compared to switch.p4

Resource	switch.p4	SQR	switch.p4 + SQR
Match Crossbar	51.56%	10.22%	61.59%
Hash Bits	32.79%	13.28%	44.75%
SRAM	29.58%	15.31%	41.35%
TCAM	32.29%	0.00%	32.29%
VLIW Actions	36.98%	6.77%	43.23%
Stateful ALUs	18.75%	15.63%	33.33%

Also, since the combined usage of all resources by switch.p4 and SQR is less than 100%, switch.p4 can easily be enhanced by incorporating SQR.

4.5 Discussion

Hardware-assisted Link Failure Detection. High-speed network cable connectors such as QSFP+ and QSFP28 "squelch" their data input/output lanes on detecting loss of input/output signal levels [167]. Modern switching ASICs are able to detect such data lane squelching and provide primitives for fast failover [139]. We investigated such hardware-assisted link failure detection in our testbed using a Barefoot Tofino switch and an Intel XXV4DACBL1M (QSFP28 to 4xSFP28) cable. We found the worst-case detection delay to be around 2.755 μs. This implies that, with hardware support, link failure detection delays are even lower, and SQR's steady-state packet buffer consumption can be further reduced.

Alternatives to on-chip Packet Buffer. An alternative way to store cached packets could be to leverage the relatively large ($\sim$ 4 GB) DRAM available on the switch CPU. However, the switch CPU's limited bandwidth on its interface to the ASIC (PCIe 3.0 x4 [33]) and its limited processing capacity make this approach infeasible for implementing SQR. This limitation is common for all switches including fixed-function [56] or partially programmable [33]. In highly congested networks where the on-chip packet buffer is a scarce resource, using expandable packet buffers implemented via DRAM and

connected directly [30, 55] or indirectly [112] to the ASIC is a better approach, since a CPU is not required to access the DRAM. Note that SQR's overall architecture still remains the same even when implemented with expandable packet buffer.

Handling Traffic Surges. SQR exploits the availability of spare buffer and egress processing from the least loaded ports dynamically. A prior measurement study has shown that high utilization and thus congestion happens on a small number of ports and not on all the ports of a switch at the same time [188]. Nevertheless, there remains a small possibility that when a switch is saturated on all ports, SQR could make the congestion worse by partially occupying the packet buffer. To address this, SQR implements a backstop mechanism that can dynamically pause packet caching (within nanoseconds) when we detect high buffer consumption, and resume only when spare buffer becomes available. With increasing adoption of delay-based congestion control protocols in datacenters [131], we expect such high buffer pressure events that can overwhelm an entire switch's packet buffer to be rare.

Deployability and Fault Tolerance. SQR runs independently on a singleton switch and thus SQR-enabled switches can be deployed incrementally in a network. A SQR-enabled switch adds link failure tolerance for each port, i.e. it can handle failures on multiple links emanating from it. Since link failures tend to be uncorrelated [73], a partial deployment of SQR-enabled switches can effectively bring down the impact of link failures. SQR will also be effective against failures such as line-card or switch failures that cause link failure detection schemes to report corresponding link failures. One limitation is that SQR will not be able to help in the event of link failures between the end hosts and the ToR switches due to the lack of alternative paths. Also, it is not designed to handle packet corruption losses. For datacenter networks, since most switches have higher availability than the links and concurrent traffic bursts on multiple switch ports [188] and concurrent link failures are rare [73], the probability of packets being lost due to simultaneous link and switch failures will be low.

Higher Link Speeds. SQR can scale to higher link speeds (25/50/100 Gbps) with an increase in buffer consumption (see equation 4.1). For a 100 Gbps port with a 30 μs link failure detection time, the worst-case steady-state buffer consumption is expected to be 375 KB. However, on average, latency-sensitive short-flows only comprise about 20% of the total bytes in typical datacenter networks [8]. Therefore, even at 100% link utilization on a 100 Gbps link, we expect SQR to handle about 20 Gbps of latency-sensitive traffic. For this average case, the worst-case steady-state buffer consumption is about 75 KB per port. When the link fails, the average case requirement of SB′+SQR spikes momentarily to 1.9 MB per port. Switching ASICs supporting 100 Gbps switches currently have around 42 MB (> 1.9 MB) of packet buffer [57]. Also, the on-chip packet buffer size for ASICs increases with supported link speeds [179]. Therefore, SQR's consumption of packet buffer can be supported comfortably by modern ASICs.

4.6 Summary

Achieving low and bounded FCTs under link failures is a step towards providing SLA guarantees on network latency in datacenter networks. We show that existing link failure management techniques fail to keep the FCTs low, as they cannot completely eliminate packet loss during link failures. By enabling caching of small number of recently transmitted packets, SQR completely masks packet loss during link failures from end-hosts. Our experiments show that SQR can reduce the tail FCT by up to 4 orders of magnitude for latency-sensitive workloads. While caching packets on the switch is an obvious idea, it is not straightforward to achieve and was not feasible until now. The significant reduction in route recovery times and increase in packet buffer sizes have made it feasible, while our design, implementation and evaluation of SQR demonstrates that it is both effective and practical. Our work suggests that on-switch packet caching would be a useful primitive for future switch ASICs.

Chapter 5

LinkGuardian: Masking Corruption Packet Losses in Datacenter Networks with Link-local Retransmission

In Chapter 4, we presented SQR which masks the impact of fail-stop link failures and prevents them from increasing the tail FCTs. SQR, however, does not work for corrupting links i.e. links with gray link failures which cause random packet losses. In this chapter, we present LinkGuardian, an in-network retransmission scheme designed to address the problem of increased tail FCTs due to gray link failures. Here, we first present 3 important trends and highlight why they make it important to handle gray link failures in the general context of datacenter networking (Section 5.2). We then present the detailed system design of LinkGuardian (Section 5.3) including its deployment strategy (Section 5.3.6) followed by results from an extensive evaluation (Section 5.4). Finally, we discuss other related issues (Section 5.5) before concluding the chapter (Section 5.6).

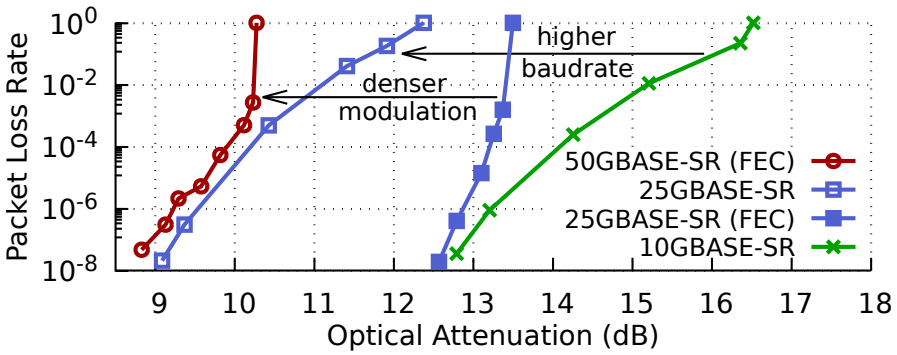


Figure 5.1: Effect of optical attenuation on high speed Ethernet standards with higher baudrates and denser modulation.

5.1 Introduction

Optical links are commonly used as switch-to-switch links in modern datacenter networks [192]. Unfortunately, optical links tend to be susceptible to data transmission errors arising from external physical factors such as physical damage, bending, or contamination due to airborne dirt particles [61, 192]. As a result, packet losses due to corruption on optical links in large warehouse-scale datacenters are common. AliBaba's recent study of hundreds of real-world service tickets showed that about 18% of the packet drops that caused network performance anomalies (NPAs) were due to packet corruption [189]. Another large-scale study across 15 Microsoft datacenters consisting of 350K optical links showed that the number of packets lost due to corruption is comparable to those lost due to congestion [192].

At the same time, Ethernet link speeds continue to increase, having increased from 25G [91] in 2016 to 400G [94] in recent years. This increase has been achieved through a combination of using multiple parallel PHY lanes, higher baudrate, and denser modulation. Figure 5.1 shows the result of a measurement experiment (details in §5.2.1) where we can see that, as the link speeds continue to increase through the use of higher baudrate (from 10G to 25G) and denser modulation (from 25G to 50G), optical links

are becoming more susceptible to optical attenuation and thus corruption packet loss.

Optical corruption can only be remedied by physically repairing the damaged links, which can take between several hours to days [192]. During this time, the impact of corruption can only be *mitigated*. The current state-of-the-art approach to mitigate corruption packet loss is to disable the corrupting links while maintaining a certain minimum network capacity [182, 192]. However, this approach is not sufficient, as it is often not feasible for some corrupting links to be disabled without violating capacity constraints. Such links will continue to cause packet drops thereby negatively impacting both throughput and latency-sensitive flows. Data from Microsoft datacenters shows that up to 15% of the corrupting links cannot be disabled under realistic capacity constraints [192].

In this paper, we apply the classical loss recovery strategy of link-local retransmission for mitigating corruption packet loss in datacenter networks. Link-local retransmission has been studied extensively [18, 19, 148] and deployed widely in wireless networks [1, 2, 88, 89]. It has desirable properties such as the recovery overheads are proportional to the corruption loss rate and localized to only the corrupting link. It can achieve sub-RTT recovery and since it is agnostic to the end-hosts, it is amenable to any transport protocol including RDMA. Yet, despite these advantages, link-local retransmissions have never been deployed in the context of datacenter networks to the best of our knowledge.

We suspect that this is because deploying link-local retransmission in datacenter networks is challenging for the following reasons: first, link-local retransmission requires packet buffering while datacenter switch buffers are generally small. The problem is further exacerbated by high link speeds that will generally require more buffering. Second, most flows in datacenter networks are short (see Figure 5.3), which increases the probability of tail packet loss. Such tail losses need to be detected and recovered at microsecond scales to provide bounded tail FCT guarantees and meet the stringent Service Level Agreements (SLAs) [51, 130, 177, 189]. Third, RDMA is being widely deployed in

modern datacenters [70, 80, 130, 190] which is more sensitive to packet reordering than TCP [84]. Therefore, packet ordering needs to be preserved while performing link-local retransmission.

In this paper, we show that, with modern programmable switches, it is now feasible to implement link-local retransmission in datacenter networks. Our system, *LinkGuardian*, can overcome the above challenges by implementing the following mechanisms: (1) a fast and efficient (low overhead) loss detection and recovery protocol to keep the recovery delay and thus the buffering requirement small (§5.3.1 and §5.3.4); (2) a novel mechanism to detect tail packet losses quickly and efficiently using a *self-replenishing* queue of "dummy packets" without the need for a timeout (§5.3.2); and (3) a "reordering buffer" at the receiver switch to maintain packet ordering along with a PFC-based *backpressure* mechanism to ensure that the buffer does not overflow (§5.3.3). While individually these techniques are relatively straightforward, our key insight is that their combination is *sufficient* to make link-local retransmission *feasible* in modern datacenter networks.

Conventional wisdom says that link-local retransmissions need to preserve packet ordering to prevent the transport layer from triggering spurious loss recovery and reduction of the sending rate [8, 11, 18, 24, 190]. We will show that in the context of datacenters, it is not *always* necessary to preserve packet ordering (§5.4.4). The key insight is that most flows in datacenter networks are short [130, 153] and most flows fit within one packet and require only 1 RTT to complete [130] (see Figure 5.3). When a flow fits within a single packet, we do not need to worry about ordering for both TCP and RDMA. For multi-packet TCP flows, out-of-order retransmission can still provide significant corruption loss mitigation for TCP flows at 100G speeds even if we cannot retransmit within TCP's reordering window. This is because even when a TCP flow spans multiple packets, it lasts only a few RTTs (flows being short). This means that if there is a corruption loss, it mostly occurs just once and thus reordering happens at most once which has minimal impact on the FCT. To this end, we show that a non-blocking vari-

ant of LinkGuardian (that implements out-of-order retransmission) not only has lower overheads but can scale better to higher link speeds (§5.4.2). However, for multi-packet RDMA flows, we currently still need to preserve ordering due to its go-back-N transport recovery.

LinkGuardian is currently implemented on an Intel Tofino switch and our testbed evaluation shows that (i) for a 100G link with a loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only 8% reduction in the link's effective link speed and requiring less than 90 KB of packet buffer; and (ii) LinkGuardian improves the 99.9^{th} percentile FCT for TCP and RDMA by 51x and 66x respectively by handling tail packet losses at sub-RTT timescales. Furthermore, LinkGuardian is complementary to existing solutions for handling corrupting links. By augmenting CorrOpt [192] with LinkGuardian, the corrupting links that cannot be disabled due to network capacity constraints can run with orders of magnitude lower loss rate without affecting application performance. By doing so, CorrOpt [192] when augmented with LinkGuardian can reduce the total loss rate in a large datacenter network by at least 4 orders of magnitude and also allow network operators to operate the network at a higher average capacity, that was not previously possible. In a network's operation, LinkGuardian lies dormant and incurs no cost until it is activated to protect a corrupting link.

The main drawback of our current implementation of LinkGuardian is that it uses recirculation for packet buffering because of hardware constraints (Tofino). With more advanced hardware like the Tofino2 [117], LinkGuardian could be realized more efficiently. We provide a sketch of how this can be done (§5.5). Nevertheless, we believe that we have made a strong case that link-local retransmission is both practical and effective for modern datacenter networks.

Loss Bucket	% Links
$[10^{-8}, 10^{-5})$	47.23%
$[10^{-5}, 10^{-4})$	18.43%
$[10^{-4}, 10^{-3})$	21.66%
$[10^{-3}+)$	12.67%
Total	100%

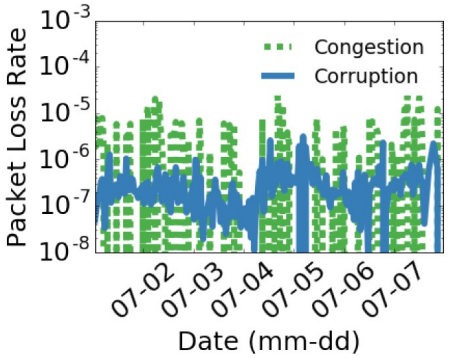


Figure 5.2: Distribution of corruption loss rates and time-varying corruption on a single link as observed by Zhuo et al. [192]

5.2 The Case for Mitigating Link Corruption

The use of optical fiber links is common in datacenter networks because they can support high data rates over longer distances (∼100 m). However, optical fiber is susceptible to bit errors due to various physical factors. While the physical factors can be varied, a majority of them lead to a single common symptom of *optical attenuation* by causing a drop in the RX optical power at the receiving transceiver [192]. This reduced RX optical power leads to decoding errors and thus corruption packet loss. In Figure 5.2, we reproduce the loss rate distribution of corrupting links as observed by Zhuo et al [192]. We also note that the loss rate on a single link can vary with time.

In this section, we present measurement studies that suggest that packet corruption cannot be ignored in datacenter networks because of (i) increasing link speeds; (ii) most flows being short; and (iii) increasing adoption of RDMA.

5.2.1 Impact of Higher Link Speeds

Ethernet link speeds have increased by a factor of more than 10 over the past 8 years. This increase has been achieved through a combination of 3 factors: (i) increase in the number of parallel PHY lanes, (ii) increase in the baud rate (symbol rate), and (iii) use

of denser modulation that packs more bits per symbol. While adding parallel PHY lanes does not change the fundamental characteristics of signal transmission, an increase in the baud rate and use of denser modulation does.

To understand the impact of higher baud rates, following the methodology of Zhuo et al. [193], we used a Variable Optical Attenuator (VOA) to add a configurable optical attenuation on an OM4 grade fiber link. We then sent standard MTU sized packets (1,518 B frames) through the optical link and measured the packet loss rates using four different configurations: (1) a pair of 10GBASE-SR transceivers [64] (10.3125 GBd); (2) a pair of 25GBASE-SR transceivers [66] that use the same modulation as 10GBASE-SR but at a higher baud rate (25.78125 GBd); (3) the same setup as (2) but with the *optional* Ethernet Reed-Solomon (RS) FEC enabled; and (4) a pair of 50GBASE-SR transceivers [65] that use a similar baudrate as 25GBASE-SR (26.5625 GBd) but a denser state-of-the-art PAM4 modulation along with the *compulsory* Ethernet RS FEC.

In Figure 5.1, we plot the packet loss rates for different levels of optical attenuation for the four different configurations. Clearly, with higher baudrate, 25GBASE-SR is more susceptible to optical packet corruption compared to 10GBASE-SR. Even with the RS FEC enabled, 25GBASE-SR performs poorly compared to 10GBASE-SR and can result in packet loss rates up to 10^{-3}. Also, with denser modulation, 50GBASE-SR is more susceptible to optical packet corruption compared to 25GBASE-SR (with similar Ethernet RS FEC enabled).

5.2.2 Most flows are short flows

In Figure 5.3, we plot the flow size distribution of several industry datacenter workloads [8, 16, 119, 154, 166]. We see that most flows are short, even shorter than the standard MTU sizes of 1500 B and 1024 B used by TCP and RDMA respectively. As a result, these flows will fit within a single packet and complete within 1 RTT under normal conditions. Except for the 2010 DCTCP web search workload [8], all other packet traces

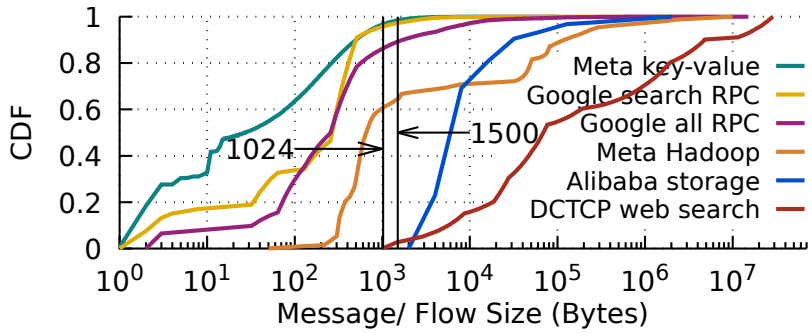


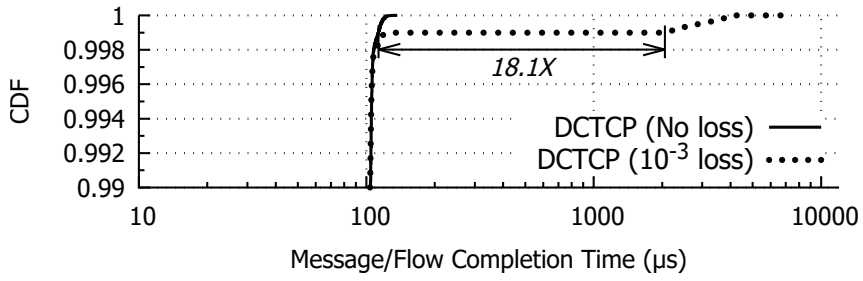
Figure 5.3: Flow size distribution of several industry datacenter workloads from 2008 to 2019 [8, 16, 119, 154, 166].

have flow sizes less than 50 KB (∼30 packets) at the 75th percentile allowing them to complete within only a few RTTs. The key takeaway is that because the flows last only a few RTTs, the additional 1 RTT delay incurred in end-to-end recovery of a corruption packet loss is expensive.

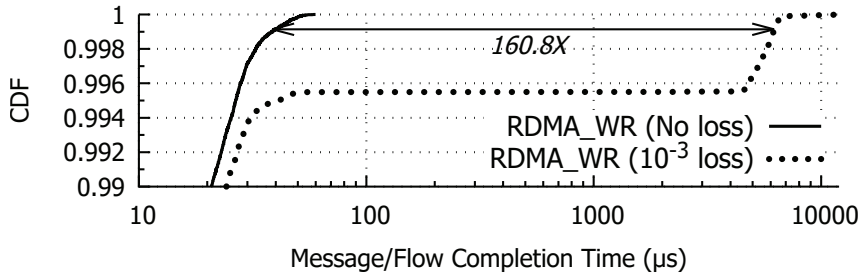
5.2.3 Impact of RDMA Workloads

With the introduction of RoCEv2 [15], the use of RDMA in datacenters networks is now becoming increasingly commonplace [70, 80, 130, 190]. Unlike traditional transport protocols, RDMA requires a lossless network fabric, which is commonly achieved through NIC-based congestion control such as DCQCN along with Ethernet flow control to prevent congestion packet loss [190]. This unfortunately makes RDMA traffic extremely vulnerable to corruption packet loss.

To illustrate the impact of corruption loss on TCP and RDMA workload, we measure the FCT of flows with 143 bytes, the most frequent flow size in Google's all RPC workload [166]. We use 25G Mellanox CX5 NICs connected through a 25G network. For TCP, RTO_{min} is set to 1 ms. For RoCEv2, we use a one-sided RDMA_WRITE operation using NIC-based reliable delivery (RC [145]) which we found to have a RTO of ∼5 ms. We chose RDMA_WRITE as it represents a shared memory write operation [187].



(a) DCTCP.



(b) RDMA WRITE.

Figure 5.4: Top 1% FCTs for 143B flows on a 25G link with and without 10^{-3} corruption packet loss.

We use a Variable Optical Attenuator (VOA) to configure 2 different packet loss rates: (i) 0 (baseline); (ii) $\sim 10^{-3}$. In all our experiments, there is no cross traffic and only a single flow exists in the system at any given time. This ensures that any performance degradation observed is solely due to corruption packet loss and not due to congestion-induced delay or loss.

In Figure 5.4, we plot the 99^{th} percentile FCTs for 300k trials, running over DCTCP and RDMA. Under no loss conditions, RDMA clearly lives up to its promise by achieving $\sim$3x lower FCT than DCTCP at the 99.9^{th} percentile. However, under 10^{-3} corruption packet loss, the FCTs for both RDMA and DCTCP degrade sharply yielding 160.8x and 18.1x higher FCT at the 99.9^{th} percentile, respectively. While it may appear that using aggressive RTO can mitigate this increase in tail FCT, there are several reasons outlined by Lim et al. [120] due to which aggressive RTO is not effective and millisecond-level

RTO remains to be the industry practice.

5.3 LinkGuardian

The corruption loss rates in real-world datacenters tend to be small (see Figure 5.2). This makes is it possible for LinkGuardian to *mitigate* the impact of corruption packet loss using link-local retransmission. To *detect* link corruption, we use a low-cost scheme that continuously monitors all optical links in the network (see Appendix A.2). Until it is activated, LinkGuardian lies dormant and imposes no cost on the network.

In this section, we provide an overview of LinkGuardian's design by describing a basic link-local retransmission (LL-ReTx) scheme, the challenges of implementing LL-ReTx at line rates, and, finally, the key ideas that make LL-ReTx practical in the context of datacenter networks.

Basic LL-ReTx. LinkGuardian can be modelled as a protocol running between a "sender" switch and a "receiver" switch (see Figure 5.5). The sender adds a monotonically increasing sequence number (seqNo)[1] to the transmitted packets and buffers a copy of the recently sent packets (in Tx buffer). These sequence numbers are used by the receiver to detect corruption packet losses. When there is no packet loss (seqNo 1-2), the receiver piggybacks the cumulative ACK information on top of reverse direction traffic (Ack2). The sender then drops the buffered copies of successfully delivered packets (seqNo 1-2). In case of a corruption packet loss (seqNo 3 in Figure 5.5), the receiver detects the gap in the sequence number when it receives the subsequent packet (seqNo 4). The receiver then sends a high-priority loss notification to the sender (Lost3) and the sender will retransmit the packet with seqNo 3 with high priority. More details on sender's packet buffering and retransmission can be found in §A.1.2.

Challenges. While this basic LL-ReTx scheme is sufficient to achieve LL-ReTX, it

[1]The sequence number is added per sender-receiver link pair.

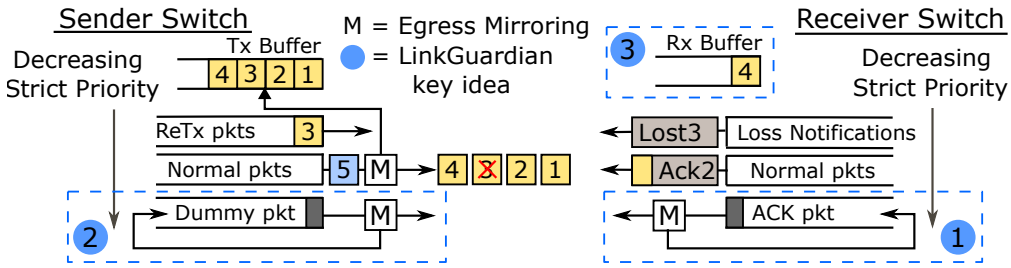


Figure 5.5: LinkGuardian Design Overview.

is not practical in a datacenter because of the following reasons:

1. **Small buffers:** Since the switches in datacenter networks have shallow buffers, the sender needs to receive the ACKs fast enough so that it can drop the buffered packets fast enough to keep the Tx buffer usage small. If we piggyback ACKs naively, they could get delayed by an arbitrary amount depending on the reverse direction traffic.
2. **Short flows:** Since most datacenter flows are short (see Figure 5.3), mostly 1 packet, it is not always possible to detect the loss of such packets based on the gap in the sequence numbers. In Figure 5.5, if the packet with seqNo 5 belonging to a short flow is lost, then the basic LL-ReTx scheme cannot detect the same until a subsequent packet (seqNo 6) is transmitted. This can lead to high-tail FCTs.
3. **RDMA flows:** The use of RDMA in datacenters networks is now becoming increasingly commonplace [70, 80, 130, 190]. Compared to TCP, RDMA performance is very sensitive to packet ordering due to the lack of a "reordering window" [84]. The basic LL-ReTx above does not preserve the original packet ordering e.g. when seqNo 3 is lost in Figure 5.5.

LinkGuardian incorporates three key ideas to address these challenges to make LL-ReTX practical in datacenter networks:

1. **Self-replenishing queue of ACK packets (§5.3.1):** LinkGuardian implements a strictly low-priority queue with one ACK packet at the receiver switch (① in Figure 5.5). This means that there will always be packets in the reverse direction even when there is no reverse direction traffic to piggyback the ACKs.

2. **Self-replenishing queue of dummy packets (§5.3.2):** LinkGuardian also implements a similar strictly low-priority queue of dummy packets at the sender switch (② in Figure 5.5). The dummy packets get sent out as soon as there is no regular traffic to allow the receiver to quickly detect tail packet losses (e.g. seqNo 5 in Figure 5.5).
3. **Reordering Buffer without Overflow (§5.3.3):** To preserve packet ordering, LinkGuardian implements a reordering buffer on the receiver (③ in Figure 5.5). A naive design would result in buffer overflow at today's datacenter lines rates. To prevent this, we use a PFC-based backpressure algorithm to throttle the sender when necessary.

Scope and assumptions. Our goal is not to completely eliminate corruption packet loss because it is too costly to achieve such a guarantee. Instead, we focus on the more modest goal of reducing the corruption packet loss rate to an operator-specified target level. To achieve the target effective loss rate, LinkGuardian also handles the case that the retransmitted copy of the packets could get lost too (§5.3.4). For the following sections, we assume that a corrupting link corrupts packets only in one direction which is the case with 91.8% of corrupting links in production [192]. However, we should highlight that handling bidirectional corruption is simply a matter of running a parallel instance of LinkGuardian in the reverse direction.

Operation modes. LinkGuardian in its *default* mode preserves packet ordering. However, we also allow running LinkGuardian in a simple mode called LinkGuardianNB, where we disable the mechanism that maintains packet ordering. Our results in §5.4.4 show that LinkGuardianNB is effective in mitigating corruption packet loss for short TCP flows because of the small flow sizes as well as TCP's support for reordering window and selective recovery.

5.3.1 Fast ACKs for minimum Buffer Overhead

In a no loss case, based on the ACK information from the receiver switch, the sender switch clears its buffer by dropping buffered packets that have been successfully received. Normally, the ACK information from the receiver is piggybacked on regular packets that are sent in the reverse direction to reduce overhead. However, by simply doing so, we cannot ensure that the ACK information is conveyed fast enough if the traffic rate on the link in the reverse direction is too low, or worse, if there is no traffic in the reverse direction.

To address this problem, we introduce a novel *self-replenishing* ACK packet queue that has a strictly lower priority compared to the normal packet queue at the receiver. The ACK packet queue is initialized with a single minimum-sized explicit ACK packet which will be sent as soon as the normal packet queue is empty. In addition, every time this ACK packet is sent, we replenish the queue by adding a new explicit ACK packet back to the same queue using egress mirroring. This is illustrated in Figure 5.5.

5.3.2 Tail Losses for Single-Packet Flows

Single-packet flows are relatively common [8, 16, 119, 154, 166]. Since losses are detected at the receiver based on the gap in the sequence number, when the packet belonging to a single-packet flow is corrupted and lost, the receiver would not detect the loss until another packet is transmitted on the link. The most common approach to detect tail losses is to employ retransmission timeouts. However, timeouts need to be set conservatively considering worst-case delays in order to avoid spurious retransmissions [120]. To eliminate the need for a timeout, we add another *self-replenishing* queue at the sender with a single "dummy" packet. This means that the sender will always have a packet to send even if there are no normal packets and the receiver will be able to detect the gap in sequence numbers even when there is corruption loss of a single-packet flow.

Algorithm 5: De-Duplication & In-Order Recovery

Applied to: protected, protected-reTx, recirculating rx-buffered pkts

```
1 if pkt.seq_no == ackNo then
2 |   forward();
3 |   ackNo = ackNo + 1;
4 else if pkt.seq_no > ackNo then
5 |   mark_pkt_as_rx_buffered();
6 |   recirculate(); // will be subjected to the algo again
7 else if pkt.seq_no < ackNo then
8 |   drop(); // de-duplication
```

5.3.3 Reordering Buffer without Overflow

To preserve packet ordering, the receiver switch will need to buffer packets whenever there is a corrupted packet until the sender receives the loss notification and successfully retransmits the lost packet. The receiver achieves this by first using recirculation to buffer the subsequent packets that arrive after a packet loss is detected. Since the packets are buffered using recirculation, they would get reordered and we need a method to ensure that the packets are forwarded in the right order after the lost packets are received from the sender. Furthermore, when more than 1 copy of the retransmitted packet is received (§5.3.4), the extra copies need to be dropped (de-duplication). We achieve these requirements by using a single state variable called `ackNo` which determines the correct next packet to be forwarded ahead. Buffered packets are continuously checked and put back into the recirculation buffer until it is their turn to be forwarded next. The pseudo-code for this is shown in Algorithm 5.

Preventing transmission stalls. In spite of our best efforts, there is still a small but non-zero probability that a retransmission will not be successful. Because we buffer packets at the receiver until all corrupted packets are received, this could stall the transmission indefinitely and cause the receiver recirculation buffer to overflow. To handle this rare but potentially fatal event, we implement a timeout at the receiver. If a retransmission does not occur within the timeout period, the receiver ignores the lost

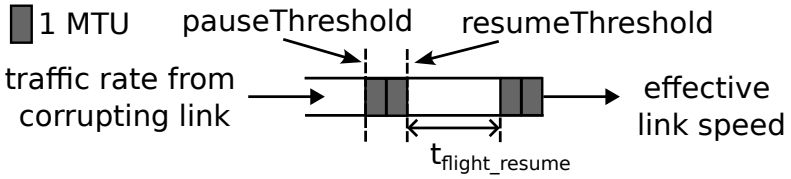


Figure 5.6: Logical view of receiver-side ingress buffer (recirculation port queue).

packet, increments the `ackNo` and continues with the remaining packet transmissions. The `ackNoTimeout` is set to a value greater than the maximum expected delay in receiving a retransmission after a packet has been found to be lost.

5.3.3.1 PFC-based Backpressure

Once a corrupted packet is detected, the receiver switch will stop forwarding packets and start buffering packets in its recirculation buffer. Because of the high link speed, there is a risk that the recirculation buffer might overflow even before the sender is notified of the packet loss and it successfully retransmits the lost packet. To avoid this problem, we employ a PFC-based pause-resume mechanism that asserts small PFC pauses on the TX MAC of the corrupting link on the sender switch. We pause only the normal packets queue (see Figure 5.5) so as not to affect the retransmission of the lost packets. The underlying principle is that we want to pause the transmission of the normal packet queue on the sender switch just enough to keep the recirculation port queue usage on the receiver switch to a small non-zero value which we set as 2 MTU (see Figure 5.6).

We note that there is a short delay before the PFC pause/resume mechanism takes effect after the receiver decides to send a pause/resume signal. Let $\texttt{t}_{\texttt{flight\_resume}}$ be the delay from when the receiver switch sends a PFC resume message to when the receiver switch receives the first packet from a previously paused regular packet queue at the sender. During this period, when the regular packet queue on the sender switch is paused, the recirculation port queue will continue to drain. The `resumeThreshold` is therefore set to a value such that during the $\texttt{t}_{\texttt{flight\_resume}}$ time, the queue will not be fully

Algorithm 6: PFC-based Backpressure

```
Input: curr_qdepth; // recirculation port's queue size
Initialization: curr_pfc_state = resume;
if curr_qdepth ¿= pauseThreshold && curr_pfc_state == resume then
    send_pfc_pause();
    curr_pfc_state = pause;
else if curr_qdepth ¡= resumeThreshold && curr_pfc_state == pause then
    send_pfc_resume();
    curr_pfc_state = resume;
```

emptied (Figure 5.6). Otherwise, the outgoing egress link at the sender will be paused excessively. Following DCQCN's recommendation [190], we set the `pauseThreshold` by leaving 2 MTU worth of space as hysteresis. The PFC pause/resume mechanism is described in Algorithm 6. The recirculation port's queue size is obtained in the dataplane on a per-packet basis. Hence, Algorithm 6 uses a flag `curr_pfc_state` to avoid sending redundant pause/resume messages.

5.3.4 Mitigating Potential ReTx Losses

If the link corruption rate is high, it is plausible that a retransmitted packet might also be lost. To improve the odds of a successful retransmission, the sender retransmits not one, but multiple copies of a buffered packet. Recall that our goal is not to completely eliminate corruption packet losses, but to reduce the loss rate to an operator-specified target level. Hence, the number of packets that are needed to be retransmitted to achieve this target with high probability is given by:

$$\text{reTx copies} = ceil\left(\frac{log_{10}(target\_loss\_rate)}{log_{10}(actual\_loss\_rate)} - 1\right) \tag{5.1}$$

For example, if a maximum loss rate of $\leq 10^{-8}$ is desired by the network operator and the loss rate on a corrupting link is 10^{-4}, then retransmitting a single copy of the buffered packet would suffice to reduce the effective loss rate to 10^{-8}. For a higher loss rate such

as 10^{-3}, 2 copies would be required. Note that the actual loss rate in Equation 5.1 is measured by a low-cost control plane based scheme (details in §A.2).

5.3.5 Implementation Details

LinkGuardian is implemented on an Intel Tofino programmable switch with about 1,800 lines of P4 code and runs entirely in the dataplane. For each packet to be protected, the sender switch adds a 3-byte LinkGuardian data header, consisting of a 16-bit `seqNo` and other metadata: the `seqNo` era and the packet type (original or retransmitted). To piggyback the updated `latestRxSeqNo` (ACK) on the reverse direction traffic, the receiver switch adds a similar 3 byte LinkGuardian ACK header. The *self-replenishing* queues of the dummy and the ACK packets are initialized by injecting a single minimum-sized packet from the switch control plane. All the state variables are maintained on a per-port basis using SRAM-based register memory.

Handling `seqNo` Wrap-around. Once LinkGuardian is activated on a link, the link is expected to carry billions if not trillions of packets in its lifetime. Therefore, any finite-sized `seqNo` would wrap around at some point. To solve the wrap-around problem, we include an additional "era bit" along with the sequence number. The era bit toggles between 0 and 1 each time the sequence number wraps around. If two numbers belong to different eras, an "era correction" is performed before comparing them. Era correction involves subtracting a constant $N/2$ from both the sequence numbers where N is the sequence number range. This era correction works correctly as long as the two sequence numbers belonging to different eras are not more than $N/2$ apart from each other.

Handling consecutive packet losses. The sender switch maintains a lookup table `reTxReqs` which is updated by the receiver and read by the sender to decide which packets to retransmit. When consecutive packets are lost, multiple entries in `reTxReqs` need to be updated simultaneously by the loss notification packet. If `reTxReqs` is implemented as a single register, such a simultaneous update is not possible due to

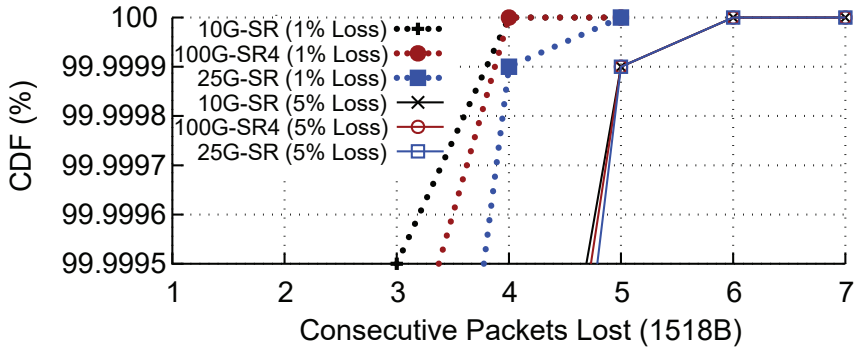


Figure 5.7: Distribution of consecutive packets lost.

hardware limitations. We had to implement `reTxReqs` across multiple 1-bit registers (details omitted for brevity) where the number of registers required is equal to the maximum number of consecutive packets lost. This number needs to be decided at compile time. In Figure 5.7, we plot the distribution of the number of consecutive packets lost that we measured by setting the VOA to induce unreasonably high loss rates of 1% and 5%. Based on Figure 5.7, our current implementation provisions for handling 5 consecutive packets lost using 5 1-bit registers.

To update the `ackNo` at the receiver when there is an ACK timeout (see §5.3.3), we follow the methodology of TimerTasks [100] where periodic packets from the switch's packet generator are used for timekeeping. In our implementation, we set the rate of these timer packets to 10 Mpps (∼1% of switch's pipeline processing capacity).

Packet Generation. To create multiple copies of a buffered packet during retransmission (in case of a high loss rate), the sender switch uses the multicast primitive. Upon detecting a loss, the receiver switch uses ingress mirroring to generate the loss notifications. Whenever PFC pause/resume packets need to be sent by the receiver, we modify the timer packets and send them to the sender switch.

Non-blocking Mode. Our prototype implementation allows us to disable LinkGuardian's functionality of preserving the packet ordering i.e. any packet lost due to corruption is simply retransmitted out-of-order. We call this mode of LinkGuardian's

operation the non-blocking mode or *LinkGuardianNB* in short. LinkGuardianNB is suitable for scenarios where a small amount of per-flow reordering does not significantly impact application performance.

5.3.6 Repairing Corrupting Links in Practice

Recall that LinkGuardian is activated on a link only when the link is found to be corrupting packets (§5.3). However, if we only enable LinkGuardian and do nothing to repair the corrupting links, then over a long period of time (∼1-2 years) nearly every link in a large datacenter network would be corrupting packets and would need to be protected by LinkGuardian. This will inflict significant overhead on all the switches as they would need to run LinkGuardian simultaneously on all the ports. Therefore, the strategy would be to deploy LinkGuardian together with CorrOpt [192]. The combined solution will function as follows. When a link is detected to be corrupting packets, we will first enable LinkGuardian on the link to immediately reduce the effective loss rate to an acceptable rate. Then we will run CorrOpt's fast checker algorithm to check if the link can be safely disabled and scheduled for maintenance without violating the network's capacity constraints. If the link can be safely disabled, then it would be disabled and scheduled for maintenance. However, if the link cannot be safely disabled, then it will continue operate with LinkGuardian enabled to ensure that there is little impact on application performance. After maintenance, whenever a link is enabled, we will run CorrOpt's optimizer algorithm to see if any of the corrupting links running LinkGuardian can be safely disabled and scheduled for repair.

By operating in this manner, both LinkGuardian and CorrOpt complement each other – LinkGuardian reduces the impact on application performance when CorrOpt fails to disable the links due to capacity constraints. This also allows CorrOpt to run at higher capacity constraints as the inability to disable corrupting links (due to high capacity constraint) no more results into high network-wide corruption losses. On the other

hand, CorrOpt helps to figure out which of the links currently running LinkGuardian could be disabled safely and scheduled for repair. Note that vanilla CorrOpt disables the corrupting links to put an immediate stop to the corruption losses. On the other hand, the combined solution of LinkGuardian and CorrOpt disables LinkGuardian-enabled links for maintenance purposes.

5.4 Evaluation

In this section, we present our evaluation results for LinkGuardian and LinkGuardianNB (out-of-order recovery). In particular, we seek to answer the following questions:

1. How effective is LinkGuardian at masking the corruption packet losses? Are we able to reduce the effective loss rate to the operator-specified target as desired? And what is the corresponding reduction in link speed?
2. How well does LinkGuardian handle tail packet loss and improve FCTs for short and single-packet flows?
3. How does LinkGuardian's performance compare with Wharf [72], the state-of-art link-local FEC solution?
4. How much buffering does LinkGuardian need and what are the associated overheads and costs of deploying LinkGuardian?
5. When deployed in a large-scale network, how effective is LinkGuardian in reducing the corruption packet loss and improving the overall network capacity?

Testbed Setup. We used the testbed setup shown in Figure 5.8, where `sw2` and `sw6` are connected by an optical fiber link that uses the OM4 grade fiber. Depending on the experiment, all switch-to-switch and host-to-switch links are either 25G or 100G. `sw2` and `sw6` act as the LinkGuardian sender and receiver respectively and we restrict their recirculation buffers to 200 KB. The link-under-test is the link between `sw2` and `sw6`. Unless otherwise stated, the loss is always introduced on the link-under-test using the

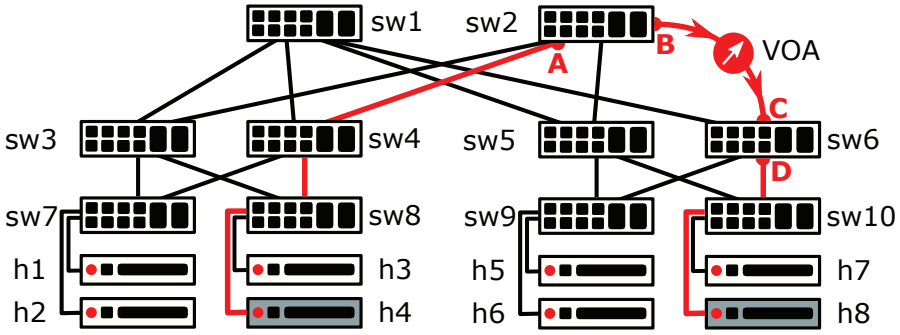


Figure 5.8: Testbed with Variable Optical Attenuator (VOA).

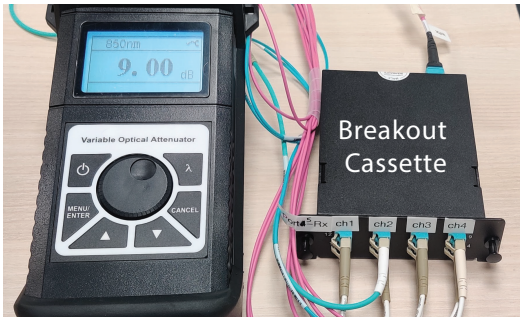
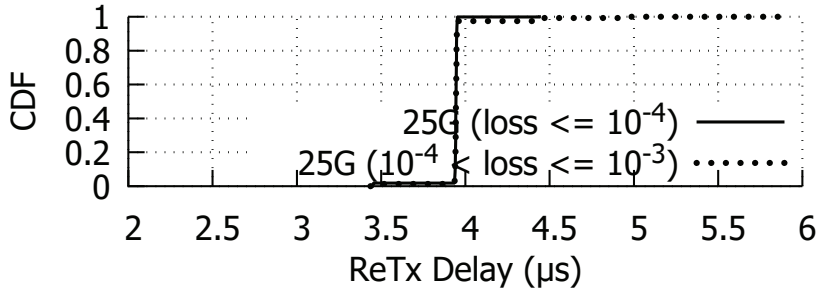


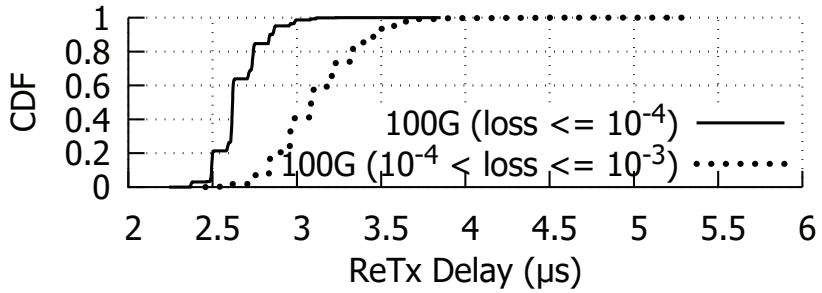
Figure 5.9: Variable Optical Attenuator (VOA) setup used in the motivation and evaluation experiments.

VOA (Figure 5.9), and loss rates are specified considering standard MTU sized packets. Following the methodology of RAIL [193], for 100GBASE-SR4 links, the VOA attenuation is applied to 1 of the 4 lanes using a breakout cassette. We set LinkGuardian's target loss rate[2] to 10^{-8} and the number of packet copies to be retransmitted is then determined by Equation 5.1 depending on the actual loss rate. We use line-rate packet generator traffic from `sw2` to measure LinkGuardian's effective link speed. Using the switch control plane, we poll the port counters for ports denoted by A, B, C and D in Figure 5.8. These counters enable us to measure the sending rate/throughput of an endpoint sender, the actual loss rate incurred due to the VOA, and the effective loss rate achieved by LinkGuardian. We also poll the queue occupancies on `sw2` and `sw6` using

[2]For MTU-sized packets, a loss rate of 10^{-8} corresponds to a bit error rate (BER) of 10^{-12} which is considered a healthy/non-corrupting link [193].



(a) 25G link speed



(b) 100G link speed

Figure 5.10: Delay observed by LinkGuardian receiver switch to receive retransmission from the time the loss was detected.

the local control plane.

The servers are equipped with Intel Xeon Silver/Gold CPUs, 128 GB memory, NVIDIA CX6-DX NICs (25G/100G) and run Linux kernel 5.4.0-91-lowlatency on Ubuntu 20.04.3. For our experiments, we use kernel-based DCTCP and NIC-based RoCEv2 (RDMA) transports. For TCP, TSO, SACK, RACK-TLP and ECN (100 KB marking threshold [53]) are enabled and RTO_{min} is set to 1 ms. The network RTT for a TCP sender is $\sim$30 μs. For RoCEv2, we use a one-sided RDMA_WRITE operation using NIC-based reliable delivery (RC [145]) which we found to have a RTO of $\sim$1 ms.

5.4.1 Parameter Tuning

LinkGuardian has three tunable parameters: `ackNoTimeout`, `resumeThreshold`, and `pauseThreshold`. In this section, we describe how we derive the appropriate values for these parameters from system parameters.

Recall that `ackNoTimeout` prevents LinkGuardian from stalling in the event that a lost packet cannot be recovered (see §5.3.3). Therefore, `ackNoTimeout` needs to be set to a value larger than the expected maximum retransmission delay. To estimate the retransmission delay, we measured the time from when the receiver switch detects packet loss to when it successfully receives the retransmission from the sender switch. Since high-priority queues are used for loss notification and retransmission, this retransmission delay is a function of the switch pipeline latencies, the link speed, and the number of retransmitted copies. If more than one copy is retransmitted, only the last copy of the retransmitted packet will be received thereby increasing the retransmission delay in the worst case.

In Figure 5.10, we plot the distribution of the retransmission delays for ∼1 million loss recoveries for standard-MTU-sized (1,518 B) packets. If the `ackNoTimeout` is set too close to the maximum retransmission delay, it can fire off prematurely and increment the `ackNo` before a retransmission is received. Hence, we conservatively set the `ackNoTimeout` to 7.5 μs and 7 μs for 25G and 100G, respectively. A larger `ackNoTimeout` leads to a slightly longer stall in transmission, but only in the unlikely event that the original packet and all the retransmitted copies are lost due to corruption.

The other parameters are `resumeThreshold` and `pauseThreshold`, which are used by the PFC-based backpressure mechanism (see §5.3.3.1). In particular, when the recirculation buffer in the sender reaches `pauseThreshold`, the sender will send the PFC pause frame; and when the buffer falls below `resumeThreshold`, it will send the PFC resume frame. Since we use a fixed hysteresis of 2 MTU, the `pauseThreshold` is

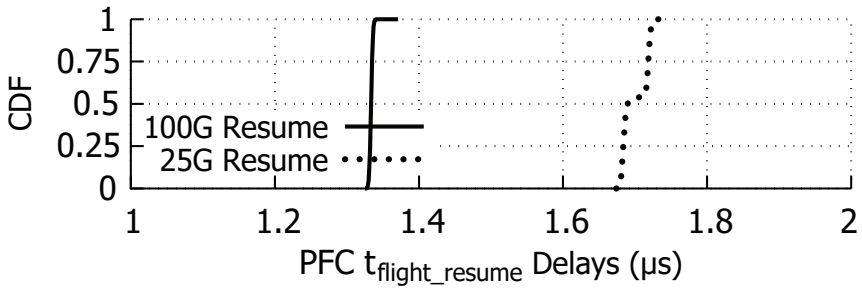


Figure 5.11: t_{flight_resume} delay observed by receiver switch.

`resumeThreshold` + 2 MTU.

If `resumeThreshold` is set too small, the receiver recirculation buffer will be empty before the sender switch successfully resumes transmissions. Hence, we set `resumeThreshold` to a value that is larger than the amount of data that would drain from the buffer during the time from when the receiver sends a PFC resume frame to when the receiver starts receiving traffic again. We refer to this time as t_{flight_resume}. t_{flight_resume} is independent of the corruption loss rate and depends only on the link speed and switch pipeline latencies.

In Figure 5.11, we plot the observed t_{flight_resume} for 25G and 100G links. In our implementation, we set `resumeThreshold` at 40 KB and 37 KB for 25G and 100G links respectively. These values correspond[3] to t_{flight_resume} values of 1.9 μs and 1.6 μs, respectively.

5.4.2 Effective Loss Rate & Link Speed

Using the packet generator on `sw2` (see Figure 5.8), we conduct a "stress test" by sending MTU-sized packets at line rate and evaluate LinkGuardian using three representative loss rates observed in production (see Figure 5.2): 10^{-5}, 10^{-4}, and 10^{-3}. As prescribed by Equation 5.1, LinkGuardian retransmits 1, 1, and 2 copies for each lost packet for these loss rates, respectively. This should theoretically result in loss rates of 10^{-10},

[3] the recirculation-based buffer drains at 100G

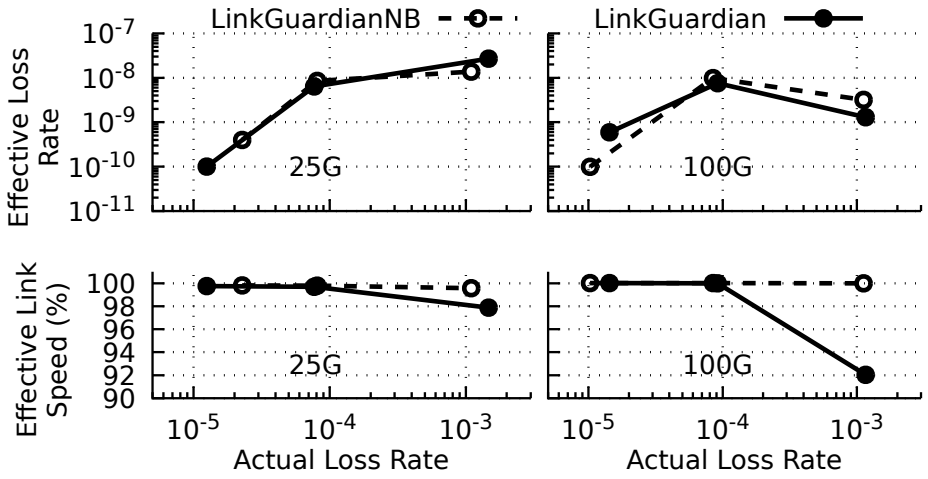


Figure 5.12: Effective loss rates achieved by LinkGuardian and the corresponding effective link speeds.

10^{-8}, and 10^{-9}, respectively. In Figure 5.12, we plot the observed (effective) loss rates achieved by LinkGuardian and the corresponding effective link speeds for 25GBASE-SR and 100GBASE-SR4 links. We observe that, except for the 25G link with 10^{-3} loss rate, the effective loss rates for both LinkGuardian and LinkGuardianNB closely match the theoretically expected loss rates. For the 25G link at the 10^{-3} loss rate, our investigations showed that the corruption losses are not independent and identically distributed (i.i.d.) and we suspect that this is the reason why the effective loss rate deviates from the theoretically expected loss rate of 10^{-9}. However, it is still very close to the target loss rate of 10^{-8}.

For effective link speed, we see that LinkGuardianNB scales much better to higher link speeds and higher loss rates compared to LinkGuardian while achieving similar effective loss rates. This is because, unlike LinkGuardian, LinkGuardianNB does not preserve packet ordering and therefore does not incur intermittent pauses in the link transmission. Nevertheless, for a 100G link with a high loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only an 8% reduction in the link's effective link speed while preserving packet ordering.

5.4.3 Impact on Transport Protocols

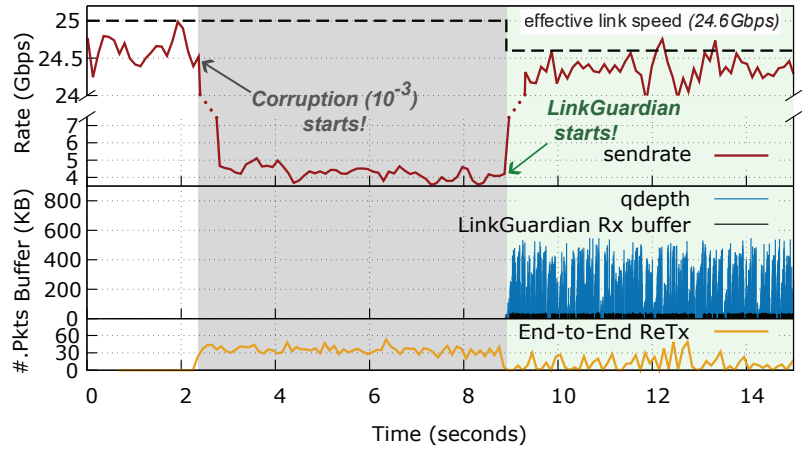
Our high-level goal is to mask the corruption packet losses from the transport layer. While we showed in §5.4.2 that LinkGuardian can reduce the effective loss rates, what matters is the net impact on transport protocols. To understand the impact of LinkGuardian, we send single flow TCP traffic from `h4` to `h8` using `iperf` with all links set to 25G. We evaluate three different TCP variants, CUBIC, DCTCP, and BBR, as they use congestion loss, ECN, and delay as congestion signals respectively. We also consider BBR to be representative of delay-based transport protocols, since the implementations for TIMELY [131] and Swift [115] were not readily available. In each experiment, we start the setup with no corruption loss. At the 2 second mark, we introduce a loss rate of 10^{-3} on the link, and approximately 5 seconds later, we enable LinkGuardian. In Figures 5.13a, 5.13b, and 5.13c, we plot the results for CUBIC, DCTCP and BBR respectively. The effective link speed in these figures is measured separately by sending a line rate UDP flow under the same experiment conditions.

CUBIC & DCTCP. In Figures 5.13a and 5.13b, we see that at 10^{-3} corruption loss, the throughputs for both CUBIC and DCTCP are reduced sharply once corruption losses are introduced. Upon enabling LinkGuardian, the corruption loses are nearly eliminated and the throughput returns to a level comparable to that before packet corruption was introduced. We also notice that there is a build-up in the flow's buffer at the sender switch (shown as "qdepth").

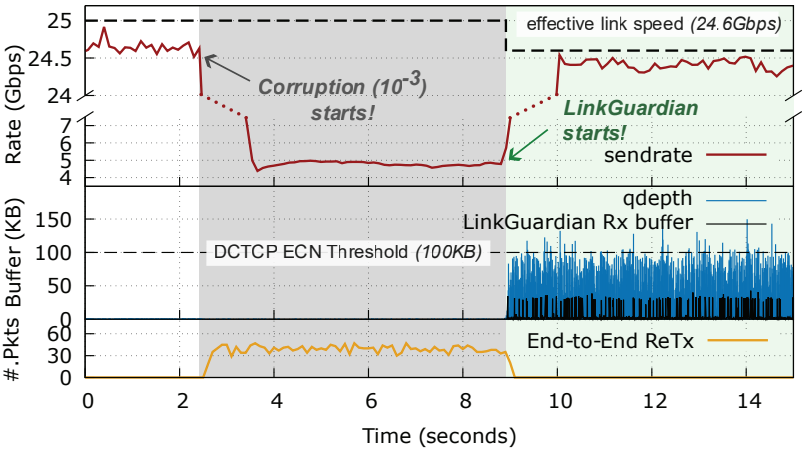
BBR. Since BBR is mostly agnostic to packet loss, we see in Figure 5.13c that it suffers minimal degradation when corruption loss in introduced[4]. Nevertheless, it seems that once LinkGuardian is enabled, we still see a small increase in the observed throughput.

Overall, we can see from Figures 5.13a, 5.13b, and 5.13c that LinkGuardian's back-

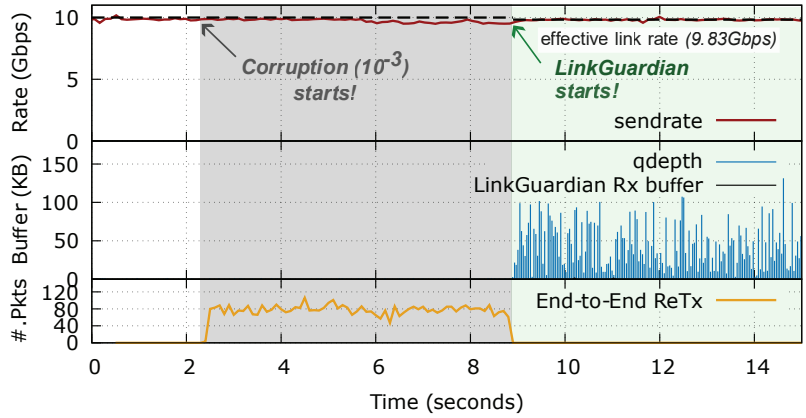
[4]We only ran BBR on a 10G link instead of a 25G link because BBR became CPU-limited when we tried to run the experiment on a 25G link, and it was not able to fully saturate the link.



(a) CUBIC on a 25G link with 10^{-3} loss.



(b) DCTCP on a 25G link with 10^{-3} loss.



(c) BBR on a 10G link with 10^{-3} loss.

Figure 5.13: Performance of LinkGuardian for CUBIC, DCTCP, and BBR Transport Protocols.

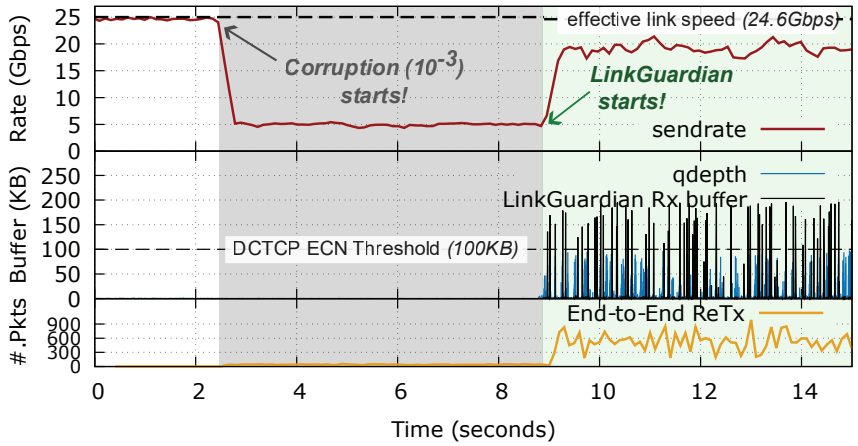


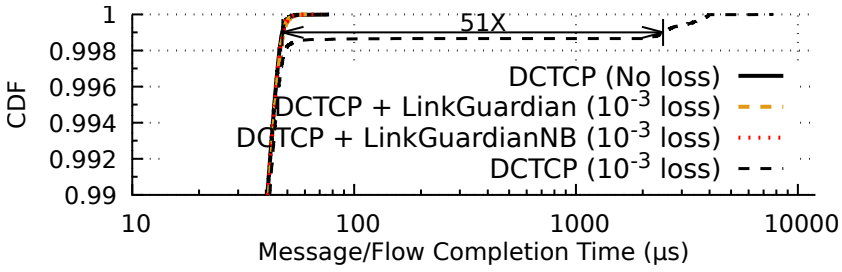
Figure 5.14: DCTCP on a 25G link with 10^{-3} loss, with PFC-based backpressure disabled.

pressure mechanism is effective at keeping its receiver-side buffer occupancy (labelled as "rx buffer") low.

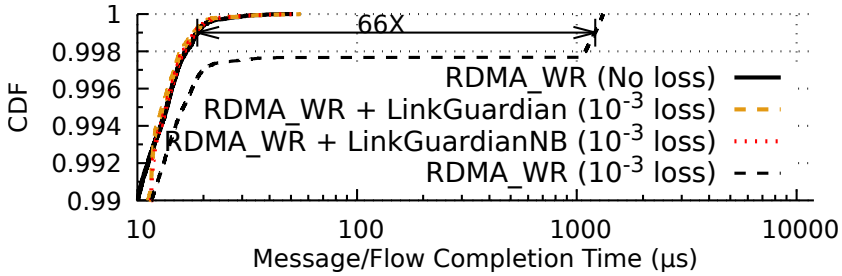
Backpressure Not Considered Optional. In Figure 5.14, we plot the results for the same experiment repeated with DCTCP, but with the PFC-based backpressure mechanism disabled. We now see a large number of end-to-end retransmissions because the recirculation buffer at the receiver periodically builds up and overflows. In fact, the end-to-end packet losses observed by DCTCP after enabling LinkGuardian are so severe that the random corruption packet losses in the period between 2 and 8 seconds are barely visible in Figure 5.14. The throughput is also lower compared to the earlier results shown in Figure 5.13b. In other words, the PFC-based backpressure mechanism is critical for ensuring that buffering at the receiver switch works as intended.

5.4.4 Tail Packet Loss and Short Flows

One-packet Flows. To evaluate how effectively LinkGuardian handles tail packet losses, we measure the FCT of 143 B DCTCP and RDMA_WR flows in our testbed with all links set to 100G while introducing a corruption loss rate of $\sim 10^{-3}$. 143 B is the most frequent flow size in the Google all RPC workload [166]. It is clear from our results in



(a) DCTCP.

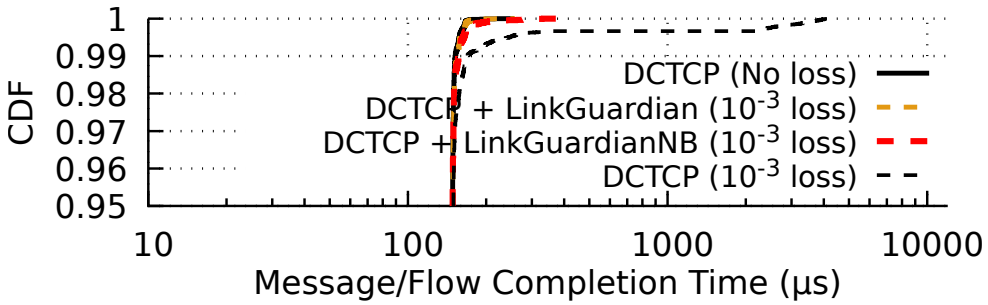


(b) RDMA WRITE

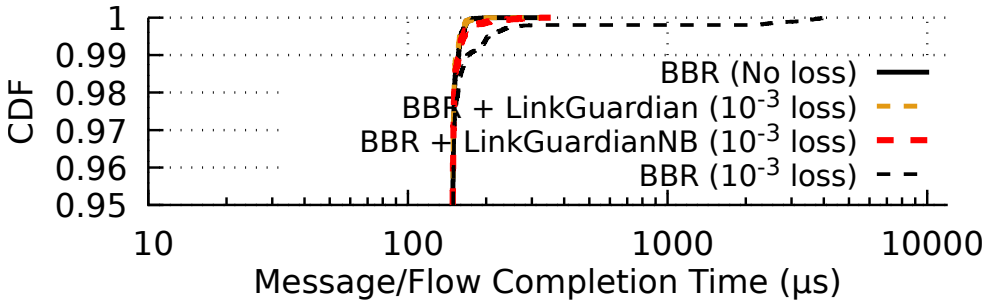
Figure 5.15: Top 1% FCTs for 143B flows on a 100G link.

Figure 5.15 that both LinkGuardian and LinkGuardianNB are able to mask the corruption losses so effectively that the performance at 10^{-3} loss rate becomes indistinguishable from the case when the link is lossless. LinkGuardian and LinkGuardianNB achieve the same performance since we do not need to worry about ordering in case of single packet flows. We note that the result in Figure 5.15 is also representative of all other flow sizes for workloads in Figure 5.3 that fit within a single packet.

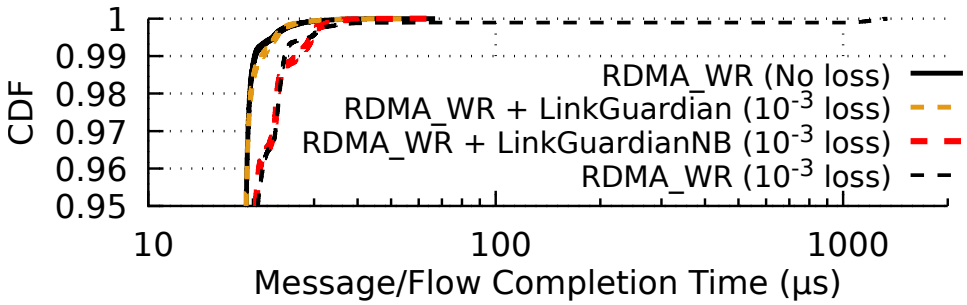
Longer (multi-packet) Flows. Next, we repeat the experiment with 24,387 B-sized flows which is the most frequent flow size in the DCTCP web search workload [8]. We plot the results when using DCTCP, BBR and RDMA_WRITE transports in Figure 5.16. We can see that the lines for LinkGuardian and no loss mostly overlap. While BBR is mostly agnostic to packet loss, this experiment shows that corruption packet loss does affect the FCTs of short BBR flows and therefore mitigating corruption loss is necessary for BBR and similar rate-based/loss-agnostic transport protocols. In Figure 5.16, we



(a) DCTCP



(b) BBR



(c) RDMA WRITE

Figure 5.16: Top 5% FCTs for 24,387B flows (17 pkts) on a 100G link.

also see that for RDMA, LinkGuardianNB provides no improvement over the loss case other than preventing RTO by handling tail packet losses. This is because RDMA's NIC-based reliable delivery has no reordering tolerance and LinkGuardianNB does not cause any reordering when it recovers the tail packet loss. On the other hand, for DCTCP and BBR, LinkGuardianNB performs nearly as well as LinkGuardian except at very high percentiles ($> 99.9^{\text{th}}$) where it performs marginally worse.

Why does LinkGuardianNB perform so well? For single-packet flows, it is unsurprising that the transport layer performance is the same for both LinkGuardian and LinkGuardianNB. For longer flows, we found that since TSO is enabled, packet bursts travel at near line rate (100G) and LinkGuardianNB is not able to perform out-of-order recovery within TCP's reordering window of 3 packets. However, since the flows are short, this does not significantly affect the FCT for two reasons: (i) corruption often happens among the last 3 packets for short flows where there is no reduction in `cwnd` as the TCP sender does not receive sufficient SACKed bytes (>= 3 MSS) while LinkGuardianNB performs a sub-RTT but out-of-order recovery; and (ii) in cases when there is `cwnd` reduction, since the flows are short, it does not significantly affect the FCT. For BBR, there is no reduction in sending rate since BBR is loss-agnostic. However, BBR still benefits from LinkGuardianNB by avoiding 1 RTT delay as well as TCP end-host stack latencies involved in end-to-end recovery.

In summary, both LinkGuardian and LinkGuardianNB improve the 99.9^{th} percentile FCT for single packet DCTCP and RDMA flows by 51x and 66x respectively. For longer flows, the 99.9^{th} percentile gains for LinkGuardian are 19x for DCTCP and BBR, and 39x for RDMA. While LinkGuardianNB performs similar to LinkGuardian for longer TCP flows (up to 99^{th} percentile), it provides little benefit in case of reordering-sensitive RDMA but does eliminate the long tail FCTs due to RTOs.

5.4.5 Contribution of different mechanisms

To understand the contributions of the different mechanisms implemented by LinkGuardian, we repeat the above experiment (24,387 B) with a variant of LinkGuardian implementing only link-local retransmission (ReTx) and then selectively enable LinkGuardian's packet order preserving (Order) and tail loss handling (Loss) mechanisms. In Table 5.1, we show the top 1% FCT results for DCTCP. Simple link-local retransmission improves the 99.9% FCT significantly as it recovers the loss of the 3rd last and the 2nd last packets in the flow which would otherwise cause an RTO due to lack of 3 MSS SACKed bytes. Additionally handling packet ordering only provides marginal gains up to 99.9%. Tail loss handling on the other hand significantly reduces FCT at all top percentiles. Notice that the two right-most columns represent LinkGuardianNB and LinkGuardian respectively, and the additional packet order preserving by LinkGuardian improves the FCT by ∼33% at 99.99% and above percentiles thereby nearly matching the performance of the no loss case. Results for BBR and RDMA (omitted for brevity) show similar trends except that for RDMA at 99.9%, ReTx+Order shows 3.75x improvement than ReTx since RDMA is more reordering intolerant compared to TCP. One may erroneously conclude that tail loss handling only helps for FCTs at 99.99% and above. However, our results in Figure 5.15 show that tail loss handling in crucial for single-packet flows.

We note here that these performance deficits exist even though RACK-TLP is enabled in our experiments. While the exact reason is under investigation, we believe that this because for very short flows RACK-TLP does not have a reliable estimate of the network RTT.

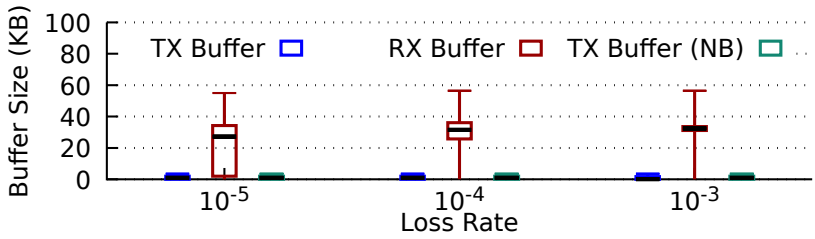
Table 5.1: Top 1% FCT (μs) for 24,387B DCTCP flows for different LinkGuardian mechanisms: tail loss handling ("Tail") and preserving packet order ("Order")

	No Loss	Loss (10^{-3})	ReTx	ReTx +Order	ReTx +Tail	ReTx+Tail +Order
99.00%	152.293	169.044	161.959	161.168	156.627	155.669
99.90%	166.877	3399.743	212.378	193.252	195.588	168.21
99.99%	197.536	4036.167	3606.115	3773.866	314.128	194.085
99.999%	253.207	4159.96	4107.404	4088.288	356.503	235.793
std dev	21.3	172.294	63.695	80.148	22.629	22.286

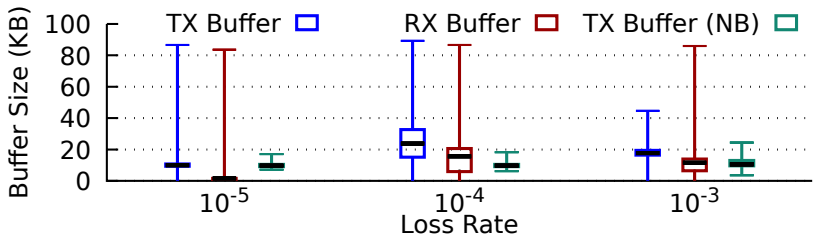
5.4.6 Overhead

In this section, we evaluate the overheads of deploying LinkGuardian. In particular, we consider 4 aspects: (i) buffer usage, (ii) protocol overhead, (iii) recirculation overhead; and (iv) dataplane resources consumed. We will show that the overheads are so low that LinkGuardian is immediately and easily deployable on modern switches. In this section, we present the overhead results corresponding to the "stress test" experiments in §5.4.2 where we run continuous line-rate traffic. These results, therefore, show the "worst case" cost of running LinkGuardian as real-world link utilization exceeds 90% only about 10% of the time [188].

Packet Buffer Usage. LinkGuardian requires packet buffer at the sender switch (TX buffer) and additionally at the receiver switch (RX buffer) when packet ordering is to be preserved. We used control plane APIs to measure the packet buffer usage which we plot in Figure 5.17 for 25G and 100G links running at three different loss rates. The key takeaway from these results is that at 25G, the TX and RX buffer usage for LinkGuardian are at most 3.6 KB ($\sim$2 MTU) and 60 KB respectively for all evaluated loss rates; at 100G, the TX and RX buffer usage are both at most 90 KB. LinkGuardianNB requires no RX buffer, while its TX buffer requirement is same as LinkGuardian at 25G and about 3x lower (24.4 KB) at 100G. This is because LinkGuardianNB has no PFC-based backpressure mechanism that could potentially delay the ACKs. To put these



(a) 25G link speed.



(b) 100G link speed.

Figure 5.17: LinkGuardian's packet buffer usage for different link speeds and packet loss rates. Whiskers show min, max, 25$^{\text{th}}$, 50$^{\text{th}}$, 75$^{\text{th}}$ percentiles.

Table 5.2: Recirculation overhead (% pipe forwarding capacity)

Loss Rate →	10^{-5}	10^{-4}	10^{-3}
25G TX	0.45	0.449	0.444
25G RX	0.661	0.662	0.664
100G TX	0.663	0.657	0.608
100G RX	0.657	0.658	0.662

numbers in context, 100G datacenter switches have 16-42 MB of packet buffer [178]. In other words, the required buffering to deploy LinkGuardian is negligible for modern switches.

Protocol Overhead. LinkGuardian adds a 3-byte header to each packet. A similar 3-byte ACK header is added to packets in the reverse direction when the ACK information needs to be piggybacked. Since standard MTU-sized frame are 1,538 octets on wire, this overhead amounts to a ∼0.2% degradation of link capacity. Note that this overhead is incurred only when LinkGuardian is activated on a link after packet corruption is detected. Both the dummy packets and explicit ACK packets do not add any overheads

on the link since they use strictly lower priority queues and thus are transmitted only when there is no regular traffic.

Recirculation Overhead. In Table 5.2, we show the recirculation overhead at both the sender and the receiver switches in terms of the percentage of the switch pipeline's processing capacity. LinkGuardianNB has the same recirculation overhead on the sender switch but zero on the receiver switch. The key takeaway is that recirculation takes up less than 1% of the switch pipeline's processing capacity, and thus the overhead is negligible for modern switches.

Dataplane Resources. LinkGuardian needs to maintain state in the dataplane on a per-port basis and uses stateful ALUs (SALUs) for stateful operations. In our current implementation, LinkGuardian requires only ∼9% of the total SRAM memory and uses ∼25% of the available SALUs. While 25% might seem high, we note that stateful ALUs are typically not used by other switch forwarding or routing functions as those perform stateless operations. Also, we believe that future switches are likely to incorporate more SALUs, while LinkGuardian will be able to support higher link speeds without the need for more SALUs.

5.4.7 Comparison with Wharf

Link-local FEC is a natural alternative to link-local retransmissions. To this end, we want to know how LinkGuardian performs compared to Wharf [72], which to the best of our knowledge, is the state-of-the-art link-local FEC to mitigate corruption packet losses. We were not able to reproduce Wharf's results experimentally because we did not have access to the required FPGA hardware. In Table 5.3, we reproduce Wharf's results *numerically* by picking the Wharf FEC parameters that gave their best reported goodput for each loss rate (c.f. Figure 8 in [72]). In our experiments, we used the same experimental setup as Giesen et al.: 10G link, TCP CUBIC, Tofino-based random packet dropping, and 4 different loss rates. Our results show that both LinkGuardian

Table 5.3: TCP CUBIC goodput (Gb/s) on a 10G Link

Loss Rate →	0	10^{-5}	10^{-4}	10^{-3}	10^{-2}
None	9.49	9.48	8.01	3.48	1.46
Wharf	n/a	9.13	9.13	9.13	7.91
LinkGuardian	9.47	9.47	9.47	9.46	9.2
LinkGuardianNB	9.47	9.47	9.47	9.46	9.2

and LinkGuardianNB compare favorably at all loss rates. In case of LinkGuardianNB, we observed that it was able to do out-of-order retransmission within TCP's reordering window for majority of times and thereby prevented the TCP sender from reducing its `cwnd` below the network BDP.

5.4.8 Effectiveness in large-scale deployment

In this section, we present the results from the simulation of a large datacenter network that runs the combined LinkGuardian + CorrOpt solution (§5.3.6). We use the same methodology that was used to evaluate CorrOpt [192] and compare vanilla CorrOpt with the combined solution of LinkGuardian and CorrOpt.

Simulation Setup. We contacted the authors of CorrOpt [192] for details on their evaluation setup. However, due to confidentiality reasons, they were unable to provide us the topology information, the link corruption traces, the simulator and the CorrOpt algorithm's implementation originally used in CorrOpt's evaluation. Therefore, we implemented a link corruption trace generator, an event-driven simulator, the CorrOpt algorithm, and the combined solution of LinkGuardian and CorrOpt in about 2800 lines of Python code. For topology, we use the state-of-the-art Facebook fabric [12] datacenter network with about 100K switch-to-switch optical links and 1:1 oversubscription ratio[5]. All switch-to-switch links are 100G and while running LinkGuardian their effective link capacity is as per Figure 5.12. For the repair time, we use the data from CorrOpt which suggests that 80% of the links are repaired in about 2 days while the remaining links

[5]supports about 500K 10G-connected or 125K 40G-connected servers

take about 4 days. Our link corruption trace generator uses the corruption loss rate and link spatial location distribution data from Microsoft's datacenters [192]. For the interarrival times of the corruption events, we use a per-link Weibull distribution with a mean-time-to-failure (MTTF) of 10K hours. The MTTF value of 10K hours is based on the reliability study of fiber links at Facebook by Meza et al. [128] (more trace generation details in Appendix A.3).

Evaluation Metrics. We use the same metrics as used by Zhuo et al. [192] to evaluate CorrOpt: **(i) Total penalty:** sum of the loss rates for all the active corrupting links in the network. **(ii) Least paths per ToR:** the fraction of paths to the spine (top) layer of the network for the worst-case top-of-rack (ToR) switch. This metric essentially captures the impact on per-ToR path diversity as corrupting links are disabled. However, since enabling LinkGuardian does not disable a link, this metric does not capture the cost of LinkGuardian which is the reduction of a link's effective capacity. To capture the same, we introduce an additional metric – **(iii) Least capacity per pod:** the total capacity of a network pod from the ToR-layer to the spine (top) layer for the worst-case pod in the network.

Recall from §5.3.6 that both the solutions disable the links subject to the network capacity constraints. The network capacity constraint is specified as the minimum fraction of paths that every ToR switch must have to the highest stage (spine layer) of the network. Figures 5.18 and 5.19 show a 1-month snapshot of the simulation result obtained using a 1-year long link corruption trace when the capacity constraint was 50% and 75% respectively. We see that the combined solution of LinkGuardian and CorrOpt reduces the total penalty by about 6 and 4 orders of magnitude for capacity constraints of 50% and 75%, respectively. We also see that with a capacity constraint of 75% (Figure 5.19), it is not possible to disable all the corrupting links at nearly all times. This is because in the Facebook fabric topology, the links between ToR and fabric switches are critical and disabling a single such link leads to the ToR switch losing 25% of its

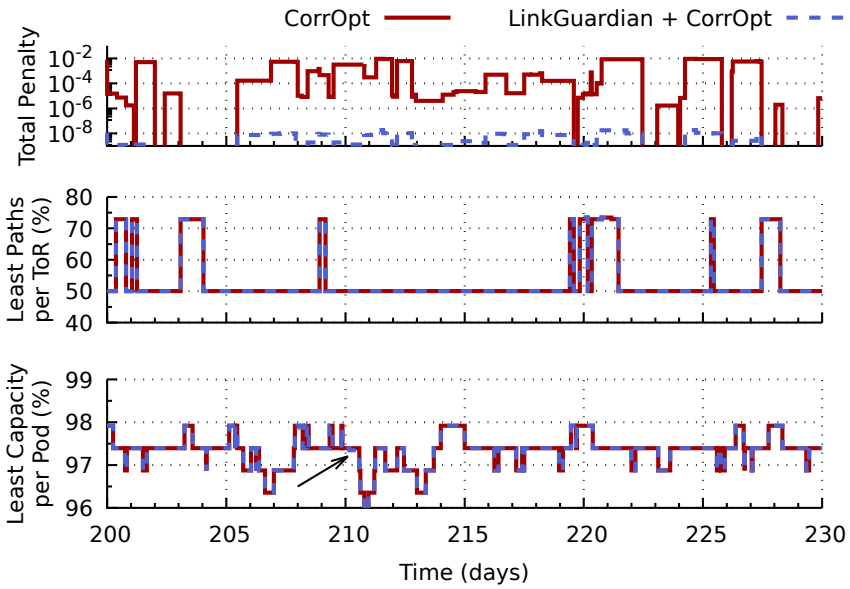


Figure 5.18: Simulation results for Facebook fabric topology (100K optical links) when the capacity constraint is 50%.

paths to the spine layer. The resulting high total penalty by CorrOpt at all times means that a network operator cannot possibly run the network at 75% capacity constraint without inflicting significant corruption packet loss on application traffic. However, with the combined LinkGuardian + CorrOpt solution, the network can be operated at 75% capacity constraint while still maintaining orders of magnitude lower total penalty.

Notice that the least paths per ToR for both the solutions go hand-in-hand since links are disabled in both the solutions using CorrOpt's algorithm. The arrow in Figure 5.18 points to the instance where the combined solution's least capacity per pod was lower than that of vanilla CorrOpt by 0.05%. This shows the small additional cost imposed by LinkGuardian in the form of reduction in the link's effective capacity that leads to reduction in the pod's capacity.

Further, to study the benefits and costs of the combined solution over the entire simulation period, in Figure 5.20 we show the CDFs of (a) the ratio of the total penalty of vanilla CorrOpt to that of the combined solution; and (b) the decrease in least capacity

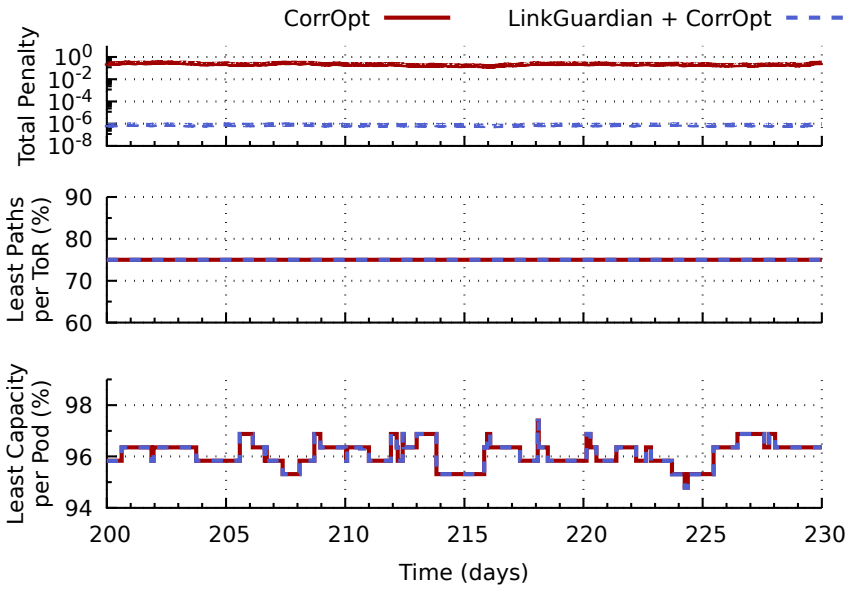


Figure 5.19: Simulation results for Facebook fabric topology (100K optical links) when the capacity constraint is 75%.

per pod by the combined solution compared to vanilla CorrOpt. In Figure 5.20a, we see that when the capacity constraint is 50%, for about 35% of the time, there is no difference in the penalty ratio as all corrupting links are disabled successfully. However, for the remaining 65% of the time and for nearly all times with 75% capacity constraint, the combined solution offers significant benefits while causing very little reduction in the pod's capacity to the core (Figure 5.20b).

Overall, compared to deploying vanilla CorrOpt, the combined solution of LinkGuardian + CorrOpt helps to keep the total penalty low when corrupting links cannot be disabled due to high capacity constraints. This also means that a network operator can now run the network at a higher capacity constraint which would have not been possible before. The additional cost imposed by the combined solution in terms of reduction in network capacity is also very low.

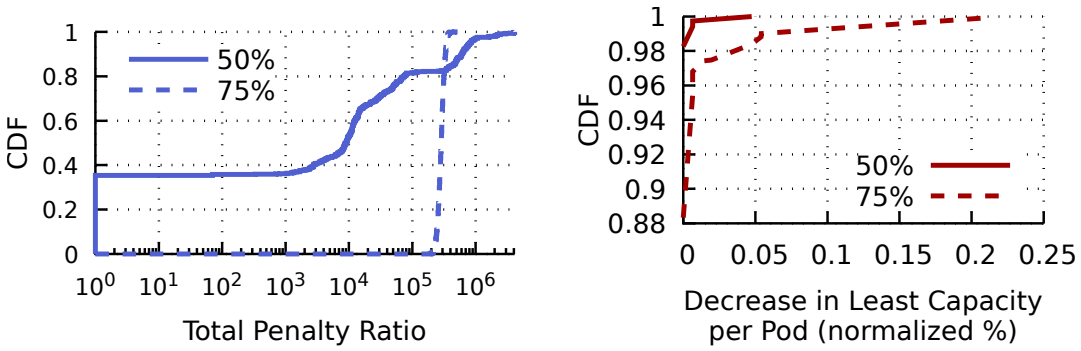


Figure 5.20: For the entire simulation period of 1 year, the CDF of (a) The ratio of total penalty of vanilla CorrOpt to that of LinkGuardian + CorrOpt; and (b) Decrease in least capacity per pod of LinkGuardian + CorrOpt compared to vanilla CorrOpt.

5.5 Discussion and Future work

In this section, we discuss a few corner cases, address the current implementation constraints with next generation programmable hardware and discuss future extensions.

Implementing LinkGuardian with Tofino2. In Figure 5.10, we see that LinkGuardian takes up to 5.25 μs to recover an MTU-sized (1,538 B on wire) packet on a 100G link. Given that it takes only about ~123 ns to serialize 1,538 bytes on a 100G link, this delay is surprisingly long. It turns out that this large delay is an artifact of our current implementation on the Intel Tofino.

Since we employ recirculation to buffer copies of recently sent packets, we cannot immediately retransmit a buffered packet as soon as a loss notification is received. In the worst case, a packet could have to cycle through the entire recirculation loop before it can be retransmitted. The same applies on the receiver switch. Recirculating basically imposes an additional and somewhat random delay.

With Tofino2, we could potentially avoid both these hardware limitations and implement LinkGuardian more efficiently. Tofino2 [117] offers new advanced flow control primitives that could be used to pause/unpause as well as achieve credit-based schedul-

ing of a queue entirely in the dataplane. These primitives could in theory allow us to implement retransmission without recirculation, but this thesis remains to be validated.

Bi-directional corruption. Currently, in our prototype, we used LinkGuardian to protect application performance from unidirectional link corruption. This suffices for most situations since 91.8% of the corrupting links have been found to be unidirectional in practice [192]. However, supporting bi-directional corruption is mostly a matter of instantiating LinkGuardian parallelly in the reverse direction. One small additional change that would be required is to increase the reliability of the control messages from the receiver switch (loss notifications, explicit ACK packets, and the PFC pause/resume) by sending multiple copies of them.

Handling multiple corrupting links on the same switch. An earlier study by Zhuo et al. reported that corrupting links exhibit weak spatial correlation i.e. corrupting links tend not to be on the same switch or topologically close [192]. Therefore, our implementation currently assumes that we only have one corrupting link per switch pipeline. A basic question that remains unanswered is the following: how can we use the recirculation port to buffer packets that come from different corrupting links? Since Tofino2 can likely implement retransmission without recirculation, Tofino2 can naturally support multiple corrupting links.

LinkGuardian and LinkGuardianNB. Our results in Figure 5.16 and Table 5.3 suggest that LinkGuardianNB could be deployed for protecting TCP flows while a complete LinkGuardian system would be required for protecting RDMA flows. Depending on the application mix and the desired level of ordering guarantees, a network operator could do a runtime configuration to run either LinkGuardian or LinkGuardianNB. In fact, while currently not implemented in our prototype, it is reasonably straightforward to allow both LinkGuardian and LinkGuardianNB to run simultaneously on a corrupting link, each protecting a different class of traffic with different ordering guarantees.

Incremental Deployment. LinkGuardian is suitable for incremental deployment

as switches are upgraded over time in a network. The links that are shared between LinkGuardian-enabled switches can then be protected by LinkGuardian if they happen to start corrupting packets. Network operators can prioritize deploying LinkGuardian at parts of the network topology where the capacity constraints are stringent or where disabling the corrupting link could impact several paths in the network. That said, a system like CorrOpt [192] would still be required as the link would need to be eventually disabled for cleaning or repair. If deployed strategically, LinkGuardian would complement CorrOpt as it would help bring down the loss rate on the links that CorrOpt is not able to disable (due to capacity constraints) as well as make it easier for CorrOpt to solve the optimization problem more efficiently.

Scalability. LinkGuardian is agnostic to the overall scale of the network as it works locally on the link between adjacent switches. The question is whether LinkGuardian would continue to work well when link speeds continue to grow ever larger. In principle, LinkGuardian would still work for higher link speeds of 400G and above. It might achieve a proportionally lower effective link speed and higher buffer overhead if the switch pipeline latency hugely dominates the retransmission delay. Based on our results in Figure 5.12, we expect LinkGuardianNB to scale better compared to LinkGuardian. However, we believe that with a Tofino2-based implementation and further dataplane optimizations, LinkGuardian should still achieve good performance with low overheads. We plan to investigate this once the hardware becomes available.

Reordering tolerance in modern transport protocols. Recently, RFC8985 [42] has introduced a new feature called the "reordering window adaptation" in the Linux TCP stack. Also, RoCEv2's NIC-based reliable transport has a new "selective repeat" feature [146] that allows more efficient selective retransmission than Go-back-N recovery. We plan to investigate the implication of these new features for LinkGuardianNB.

5.6 Summary

In this chapter, we present LinkGuardian which uses link-local retransmission to mitigate corruption packet loss in datacenter networks. While the basic idea is straightforward, to the best of our knowledge, we are the first to validate that a combination of simple techniques can make link-local retransmission practical in modern datacenter networks. LinkGuardian is able to recover from tail packet losses efficiently at sub-RTT timescales, and is, therefore, able to keep FCTs low and avoid timeouts. With a configurable target loss rate, LinkGuardian will allow network operators to work with corrupting links with moderate loss rates (between 10^{-3} and 10^{-5}) like healthy links at a marginally reduced link speed with little overhead. We also propose a combined solution of LinkGuardian with CorrOpt that is able to reduce the effective loss rate throughout the network when corrupting links cannot be disabled due to capacity constraints. It also allows network operators to run the network at much higher capacity constraints as the impact of the failed-to-disable corrupting links is now significantly reduced by LinkGuardian. Overall, we believe that we have made a strong case that link-local retransmission is both practical and effective for modern datacenter networks.

Chapter 6

Conclusion and Future Directions

With the increasing push of businesses and services to the cloud, modern datacenter networks continue to grow both in their scale and complexity. At the same time, emergence of new low-latency interactive applications such as AR/VR are making the SLA requirements from datacenter networks even more stringent. Therefore, handling network link failures as well as unexpected congestion events is crucial to ensuring that datacenter networks are able to consistently meet such stringent SLAs. In this thesis, we propose novel in-network techniques that mitigate the impact of network link failures on application performance and also provide high-resolution monitoring to tackle unexpected congestion events.

In this chapter, we first discuss the future directions including recommendations for future programmable hardware, as well as scalability and adoption issues for the proposed solutions. We then conclude the chapter and this thesis by summarizing our contributions in the broad context of datacenter networking and cloud infrastructure.

6.1 Future Directions

6.1.1 Temporal packet buffering beyond handling link failures

The key design aspect of both SQR and LinkGuardian is the temporal buffering of packet copies of recently sent packets. Temporal buffering of packet copies is a useful building block which can be used in applications beyond handling link failures. For example, it could be used to protect certain highly critical packets against congestion packet loss. A switch transmitting out a highly critical packet would make its copy and buffer the same. If the next-hop switch transmits the packet successfully, it would inform the previous switch that this packet was transmitted successfully and the previous switch can then drop the buffered copy of the packet. In case the next-hop switch drops the critical packet, it would inform the previous switch that the packet was dropped due to congestion, and the previous switch would then retransmit the critical packet. Repeating this scheme between every consecutive pair of switches, highly critical packets could be protected against congestion loss in addition to link failures.

6.1.2 Better dataplane primitives for temporal packet buffering

In this thesis, we achieved the required temporal buffering for SQR and LinkGuardian through the recirculation primitive action available in today's dataplane programmable switches. Recirculation, however, incurs unnecessary overhead in terms of the pipeline processing capacity and adds latency in addition to causing reordering of packets. Ordering the packets back again costs additional recirculation which again adds more latency. Newer generation programmable switches such as Tofino2 provide primitives such as Advanced Flow Control (AFC) that allow pausing and resuming egress port queues in the dataplane [117]. We studied AFC in details and found out that using AFC can help to prevent reordering of the temporally buffered packets. However, it cannot completely

eliminate the need for recirculation. The crux of the issue here is that the packet buffer memory is only available for use in the form of egress port queues. This means that when we buffer the packets, we should already know their potential next-hop destination, which may not always be the case (e.g. the LinkGuardian receiver switch). Further, to perform any meaningful operations on the buffered packets, these packets need to enter the egress pipeline and if for some reason, they were to be buffered again or sent out on another egress port, we need to recirculate them. Overall, what this means is that, despite of new dataplane primitives such as AFC, there is still a need for better dataplane primitives to support temporal packet buffering.

Based on our experience, we recommend two primitives, that if made available, could make temporal packet buffering more efficient and eliminate the need for recirculation. First, a primitive that allows to specify an identifier, a timeout and a timeout action for a cloned (mirrored) copy of a recently sent packet. The timeout action could either be drop or transmit to a specific egress port queue. Second, another primitive that allows to specify an identifier, and an immediate action for an already buffered packet. The allowed immediate actions are the same as the timeout actions. For pipelined architecture switches, the execution for these two primitives would have to be mainly done by the buffering and queuing engine (BQE) of a switch dataplane and the specification of the different parameters would have to be done through the forwarding pipeline's metadata. The implementation feasibility of these two primitives for pipelined architecture switches remains a matter of further investigation. However, we believe that these two primitives are definitely feasible to achieve on non-pipelined programmable switch platforms such as Juniper's Trio [183].

6.1.3 Fast and Efficient Monitoring of Link Failures

In this thesis, we propose SQR and LinkGuardian to mitigate the impact of link failures on application performance. Both SQR and LinkGuardian provide mitigation *after* the

link failure has been detected. For detecting fail-stop link failures, SQR relies on existing methods [122, 133]. The drawback with the existing methods is that they are slow (10's of μs to 100's of ms) in terms of the time they take to detect link failures. Additionally, for detecting gray link failures, a sizeable amount of application traffic needs to suffer corruption packet drops before the link can be designated as having a gray failure with a certain corruption loss rate. Several existing works make an assumption that fail-stop link failures could be detected on the orders of microseconds using network transceiver features such as Tx/Rx Squelch [167] together with hardware support from the switch dataplane. However, to the best of our knowledge, the Tx/Rx Squelch [167] feature of transceivers has not been validated and there exists no study reporting its fidelity in practice. Similarly, for detecting gray link failures, there does not exist a solution that minimizes the impact on application traffic. Therefore, there is certainly a scope to conduct an in-depth study of existing link failure detection techniques – both fail-stop and gray – and subsequently develop a more efficient solution that minimizes the penalty to the application traffic.

6.1.4 Scaling to future link speeds

Within the next decade, we expect the Ethernet link speeds of 400G and 800G (colloquially known as "Terabit Ethernet") to become common place in datacenter networks. An important question therefore is – whether the proposed in-network techniques in this thesis would scale to these future link speeds.

BurstRadar should have no problem scaling to higher link speeds. This is because when link speeds become higher, the dataplane pipelines also become proportionally faster. Also, as has been the case so far, the support for egress mirroring (required by BurstRadar for courier packets) also scales with port speeds. For similar reasons, SQR's main technique should also have no problem scaling to higher link speeds. The main challenge that SQR will face in scaling to higher link speeds is in keeping its

overheads low. If future generation switches support the new primitives mentioned in Section 6.1.2, then that would help immensely in reducing the recirculation overhead and the accompanying latency. However, larger link speeds would also mean higher packet buffer requirement for SQR. While faster and efficient fail-stop failure detection techniques (Section 6.1.3) would certainly reduce the buffer requirement, we do not expect the reduction to be significant. This means that SQR may not be practical for switches with high-link speeds and shallow buffers. However, it is likely that SQR will still remain practical if implemented on switches such as the Juniper Trio [183] that offer a large extended packet buffer (on the order of a few GBs). Such extended packet buffers are typically implemented using memory technologies such DRAM which have a higher latency. However, since fail-stop link failures and the subsequent retransmission by SQR is a one-off event, higher latency to access the extended packet buffer is not a big concern for SQR.

In case of LinkGuardian, scaling to higher link speeds poses challenges both in terms of performance as well as overheads. From Figure 5.12, we see that, for the same loss rate, as the link speed becomes higher, the degradation in LinkGuardian's effective link capacity becomes larger. This is because, for higher link speeds, the pipeline latencies of the sender and receiver LinkGuardian switches start to dominate the total retransmission delay. Since LinkGuardian performs in-order retransmission by default, each time there is a corruption packet loss, the transmission on the link needs to halt for the retransmission delay amount of time. As a result, we do not expect LinkGuardian's effective link capacity to scale very well with higher link speeds when the corruption loss rates are high ($>=10^{-3}$). However, for majority of times, real-world corruption loss rates are less than 10^{-3} (see Figure 5.2). Furthermore, the new primitives mentioned in Section 6.1.2 would certainly help lower the degradation in effective link capacity of LinkGuardian by reducing the retransmission delay. Also, based on the results in Figure 5.12, we can expect LinkGuardian's non-blocking mode (LinkGuardianNB) to fare well at higher

link speeds since it does not stall the transmission on the link. While in Figure 5.16c, we see that LinkGuardianNB does not work well with RDMA, we expect this issue to be resolved in the near future with later generation of RDMA NICs [144]. As for the performance of TCP with LinkGuardianNB at higher percentiles ($> 99.9^{\text{th}}$), we believe that there is scope for performance improvements in the end-host TCP stack. Overall, for LinkGuardian at future link speeds, we believe that LinkGuardianNB will be a more practical solution when performance improvements are also made at the end-point transport stacks.

6.1.5 Adoption in practice

We believe that the research presented in this thesis is translational and can be adopted for practical deployments. Both BurstRadar and SQR run on a singleton switch requiring no coordination with any other switches. Therefore, these two systems are the easiest to adopt for practical deployments. A system called DTEL [110] from Intel (previously Barefoot Networks) implements nearly the same snapshot algorithm as BurstRadar[1]. DTEL currently ships as a part of Intel's reference `switch.p4` implementation. Unlike BurstRadar, deployment of SQR, however, is contingent on already existing support on the switch for link failure detection and backup path selection. Deploying SQR would therefore need more work as SQR would need to be integrated with the existing failure detection and backup path selection mechanisms on the switch.

LinkGuardian implements a protocol between a pair of switches and is therefore not so straightforward to deploy compared to BurstRadar or SQR. As discussed in Section 6.1.4, the non-blocking version of LinkGuardian (LinkGuardianNB) is expected to scale better for future link speeds. LinkGuardianNB is also much simpler in its design and implementation. If the end-point transport's performance issues when working with LinkGuardianNB are addressed in the future (Section 6.1.4), we can expect

[1]We do not claim that DTEL was inspired by BurstRadar. To the best of our knowledge, DTEL and BurstRadar were developed independently.

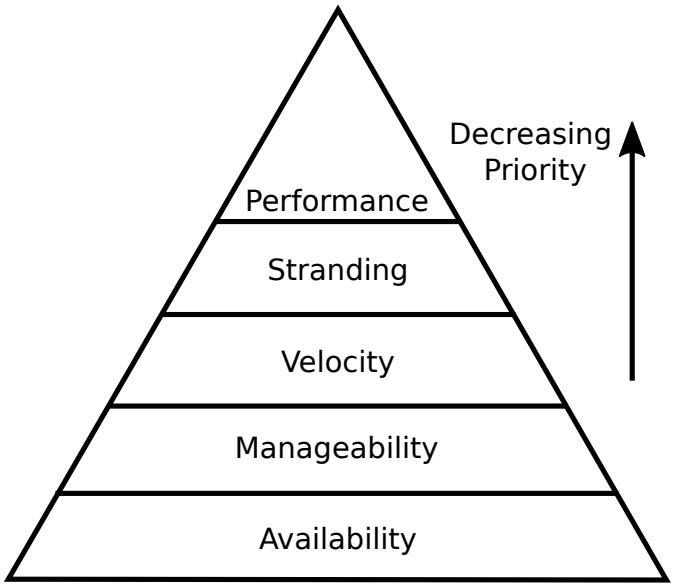


Figure 6.1: Priority order for infrastructure work at Google Cloud [172]

LinkGuardianNB to have a lower resistance for adoption. Our P4 implementation for LinkGuardianNB defines the different messages types used in the protocol between the adjacent switches. With sufficient traction, LinkGuardianNB protocol could be formalized into an Internet RFC along with the P4-specified packet header formats. For compatibility check, the protocol could include an additional handshake step where the control plane of a switch would check for LinkGuardianNB support with its adjacent switches. If supported, the adjacent pair of switches would setup the necessary initial state before activating the protocol in the dataplane.

6.2 Summary of Thesis Contributions

Since the advent of datacenter networks, there has been a significant amount of work focusing on performance [6, 8, 9, 10, 17, 76, 83, 87, 149, 173, 180] and manageability [75, 141]. Figure 6.1 shows the priority order for infrastructure work at Google Cloud [172]. We see that availability forms the highest priority for large cloud operators

without which none of the other aspects of a datacenter network can be guaranteed. In datacenter infrastructure, there has been a long standing-tradition of building reliable systems on top of cheaper and unreliable components in order to achieve a good cost vs. reliability tradeoff. Examples include building large-scale storage systems on top of inexpensive commodity disks [71]. Constant monitoring, error detection, and automatic recovery are integral to such systems. In the same spirit, this thesis contributes in-network techniques that provide monitoring, error detection and automatic recovery for transient congestion and hardware failure events occurring at individual links in a datacenter network. Essentially, by masking datacenter network link faults from applications' reliability metrics such as the tail FCTs, the proposed in-network techniques enable datacenter network operators to better handle the cost vs. reliability trade-off.

As link speeds keep increasing and the availability and performance demands of applications also increase, network management problems (including failures) would be required to be solved in real time [59]. Essentially, datacenter networks should become self-driving/self-patching networks [58, 59] where network faults patch by themselves allowing the network to run seamlessly with minimal impact on the reliability (SLAs). This thesis therefore represents a contribution towards this vision.

Appendices

Appendix A

LinkGuardian

A.1 Protocol Details

In this Appendix, we provide some details that might be helpful for understanding our implementation of LinkGuardian, but which are not essential for understanding the key ideas and contributions of our work.

A.1.1 Loss Detection & Notification

In Figure A.1, we list the state variables maintained by the sender and receiver switches and the different packets that are exchanged. The sender maintains a monotonically increasing `seqNo` while the receiver records the latest received `seqNo` as `latestRxSeqNo`.

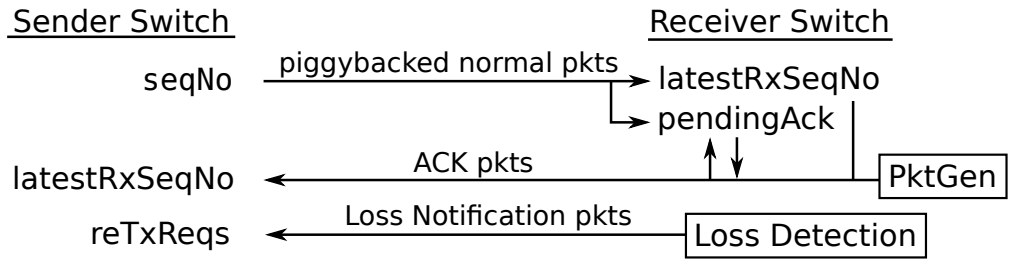


Figure A.1: State maintained by LinkGuardian switches and different types of packets that read/update it.

A copy of the `latestRxSeqNo` is also maintained at the sender, which the receiver keeps updating. The sender also maintains a lookup table called `reTxReqs`, which records the sequence numbers of the packets for which retransmission is requested.

For each packet that is transmitted on the corrupting link (protected packet), the sender adds the `seqNo` to the packet (using a custom header) and increments it by 1. The sender uses egress mirroring to also make a copy of the packet along with the added sequence number and buffers it until the receiver notifies that the packet was received successfully. On the receiver, when a protected packet is received, it updates the `latestRxSeqNo` to the `seqNo` in the packet and also sets the `pendingAck` to 1. `pendingAck` set to 1 denotes that the copy of `latestRxSeqNo` on the sender is yet to be updated.

No Loss Scenario. When there are no corruption packet losses, the `latestRxSeqNo` on the receiver would increase by 1, each time a protected packet is received. On every update of the `latestRxSeqNo`, the receiver must update the `latestRxSeqNo` on the sender as soon as possible so that the sender can drop the buffered packets that are successfully delivered. This timely update of the `latestRxSeqNo` on the sender is critical to ensure that LinkGuardian's use of the packet buffer at the sender is kept to a minimum.

Loss Scenario. When a protected packet(s) gets corrupted and dropped by the receiving MAC, the receiver observes that the `latestRxSeqNo` is incremented by more than 1. On noticing this, the receiver activates a `LossDetection()` routine. In this routine, the receiver generates a new packet called "Loss Notification" which contains information about the missing sequence number as well as the `latestRxSeqNo`. This loss notification packet is sent to the sender through a high-priority queue (see Figure 5.5) to ensure timely recovery. On reaching the sender, the lookup table `reTxReqs` (Figure A.1) is updated with the sequence numbers of the packets that need to be retransmitted.

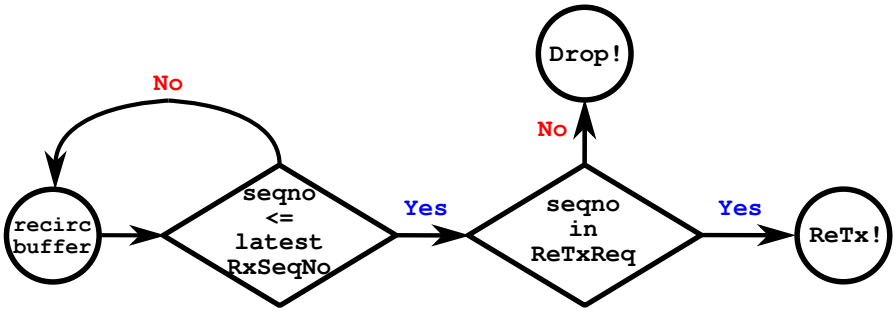


Figure A.2: Sender-side buffering and Retransmission.

A.1.2 Sender-side Buffering & Retransmission

For each packet that is sent on the corrupting link, the sender switch adds a monotonically increasing `seqNo` and uses egress mirroring to create a copy of the packet for buffering. The packet buffering on the sender switch is realized through recirculation. Specifically, the buffered copy of the protected packet is sent to the recirculation port of the switch dataplane pipeline. At the same time, as described in §A.1.1, the receiver switch keeps the `latestRxSeqNo` on the sender switch updated and additionally updates the lookup table `reTxReqs` in case of a corruption packet loss. Each time the buffered packet completes a recirculation loop, the sender switch applies the logic shown in Figure A.2 to the packet's sequence number. Essentially, if the buffered packet's sequence number is less than or equal to the `latestRxSeqNo`, the sender switch checks the `reTxReqs` lookup table to see if a retransmission is requested for that sequence number. If so, the packet is retransmitted through a high-priority queue (see Figure 5.5) or the packet is dropped otherwise. If a packet is retransmitted, its sequence number is cleared in the `reTxReqs` table. If the buffered packet's sequence number is greater than the `latestRxSeqNo`, then we do not know yet if the packet was successfully received or not and therefore the sender switch continues to buffer the packet through recirculation.

A.2 Monitoring Links for Corruption

To detect corrupting links, we implemented `corruptd`, a daemon which runs at the local control plane of the programmable switches.

Detecting Corrupting Links. `corruptd` periodically polls the driver (in this chapter, we configure the interval as 1 second) to extract the switch port RX statistics, specifically, `framesrxok` and `framesrxall`. We maintain a moving window of 100M frames to compute the link loss rates, given by $L = \frac{framesrxok}{framesrxall}$. When $L \geq 10^{-8}$ for any particular link, the upstream transmitting switch will be notified to activate LinkGuardian.

Notification and Activation. For scalability, `corruptd` daemons communicate through a publish-subscribe (PubSub) pattern using Redis. Each daemon subscribes to link corruption notifications relating to the local switch's links. Upon receipt of a notification, `corruptd` pushes corresponding data plane match-action table entries to activate LinkGuardian for the corrupting link depending on the target and the actual loss rates (see Equation 5.1).

A.3 Link Corruption Trace Generation

A link corruption trace is essentially a time series of link corruption events where a link corruption event denotes which link started to corrupt packets and at what loss rate. To determine the time at which a link would start corrupting packets, we assume a per-link 1-parameter Weibull distribution with a constant shape parameter (β). This is because the location parameter of the Weibull distribution (γ) is zero since it is not guaranteed that all links in a large warehouse-scale datacenter would not start corrupting packets during a certain initial period. Also, the shape parameter (β) is equal to 1, since the corruption is purely caused by random external events such a connector contamination,

fiber bending, etc. Therefore, the per-link Weibull PDF that determines the time until a link's next failure is given by

$$f(t) = \frac{1}{\eta} \times e^{-\left(\frac{t}{\eta}\right)} \tag{A.1}$$

where the parameter η is the mean-time-to-failure (MTTF) of a link. A study by Meza et al. [128] showed that for fibers links from different vendors considered in their study, the mean time between the link faults was at most 10,000 hours. We conservatively use the value of 10,000 hours as the MTTF (η in Equation A.1) since Meza et al. did not specifically consider only intra-datacenter links. What this means is that on average, it would take 10,000 hours (or 1.15 years) for a fiber link to start corrupting packets from the time it was last repaired.

To generate the trace, we first draw samples from the Weibull distribution independently for each link to determine the times at which each link would start corrupting packets. This gives us the various times of the corruption events and the link involved in each corruption event. Then for each corruption event, we use the corruption loss rate distribution from CorrOpt (c.f. Table 1 in [192]) to determine the loss rate. This list of corruption events sorted by time forms the link corruption trace. We note that the trace generated using the above methodology has a nearly random spatial distribution of simultaneously corrupting links which matches the observation by Zhuo et al. [192] in production datacenters.

Bibliography

[1] 3GPP. 2007. TS 36.321: E-UTRA; Medium Access Protocol Specification (Release 8). (2007).

[2] 3GPP. 2020. TS 36.321: LTE; E-UTRA; Medium Access Protocol Specification (Release 16). (2020).

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. 2009. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of SC*.

[4] Akamai. 2017. Akamai Online Retail Performance Report. (2017). Retrieved 2022-04-06 from `https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report`

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Procedings of SIGCOMM*.

[6] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of SIGCOMM*.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of SIGCOMM*.

[8] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of SIGCOMM*.

[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of NSDI*.

[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of SIGCOMM*.

[11] Mark Allman, Vern Paxson, and Ethan Blanton. 2009. TCP Congestion Control. *RFC 5681* (2009).

[12] Alexey Andreyev. [n. d.]. Introducing data center fabric, the next-generation Facebook data center network. ([n. d.]). `https://bit.ly/3uvNlcQ`.

[13] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-SpeedNICs. In *Proceedings of NSDI*.

[14] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *Proceedings of NSDI*.

[15] InfiniBand Trade Association et al. 2014. RoCEv2 Architecture Specification. (2014).

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of SIGMETRICS*.

[17] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of NSDI*.

[18] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. 1996. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of SIGCOMM*.

[19] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. 1995. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of MOBICOM*.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. 60, 4 (2017).

[21] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition.* Vol. 13. Morgan & Claypool Publishers.

[22] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic in-band network telemetry. In *Proceedings of SIGCOMM*.

[23] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of IMC*.

[24] Ethan Blanton, Mark Allman, Lili Wang, Ilpo Jarvinen, Markku Kojo, and Yoshifumi Nishida. 2012. A conservative loss recovery algorithm based on selective acknowledgment (SACK) for TCP. *RFC 6675* (2012).

[25] David Borman, B Braden, Van Jacobson, and R Scheffenegger. 2014. Protection Against Wrapped Sequences. IETF RFC 7323. (2014). `https://tools.ietf.org/html/rfc7323#section-5`.

[26] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *SIGCOMM CCR* 44, 3 (2014), 87–95.

[27] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Mar-

tin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of SIGCOMM*.

[28] Broadcom. 2010. Trident+ Buffer Size. (2010). `https://goo.gl/9LUHWa`

[29] Broadcom. 2013. Trident II Buffer Size. (2013). `https://goo.gl/3eWY7T`

[30] Broadcom. 2016. StrataDNX Qumran-AX Ethernet Switch Series. (2016). `https://bit.ly/2K1GYYR`

[31] Broadcom. 2016. Tomahawk+ Buffer Size. (2016). `https://goo.gl/3eWY7T`

[32] Broadcom. 2017. Tomahawk II Buffer Size. (2017). `https://goo.gl/3eWY7T`

[33] Broadcom. 2018. Trident 3 Ethernet Switch Series. (2018). `https://bit.ly/2HBgKut`

[34] Jake Brutlag. [n. d.]. Speed Matters. ([n. d.]). `http://ai.googleblog.com/2009/06/speed-matters.html`

[35] Carmelo Cascone, Davide Sanvito, Luca Pollini, Antonio Capone, and Brunilde Sansò. 2017. Fast failure detection and recovery in SDN with stateful data plane. *International Journal of Network Management* 27, 2 (2017), e1957.

[36] Cavium. 2018. XPliant Ethernet Switch Product Family. (2018). `https://goo.gl/xzfLLo`

[37] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Luo, Yongqiang Xiong, Xiaoliang Wang, et al. 2016. Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers. In *Proceddings of ATC*.

[38] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luo, Yongqiang Xiong, Xiaoliang Wang, et al. 2016. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *Proceedings of NSDI*.

[39] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rot-

tenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*.

[40] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of WREN*.

[41] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. 2014. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center. In *Procedings of NSDI*.

[42] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. *RFC 8985* (2021).

[43] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of SIGCOMM*.

[44] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. 2013. Increasing TCP's Initial Window. IETF RFC 6928. (2013). `https://tools.ietf.org/html/rfc6928`.

[45] Cisco. 2017. Monitor Microbursts on Cisco Nexus 5600 Platform and Cisco Nexus 6000 Series Switches. (2017). `https://goo.gl/5Xxhpm`

[46] Cisco. 2019. Configuring QoS - Catalyst 3850. (2019). `https://bit.ly/2W6jOTo`

[47] Benoit Claise. 2004. Cisco Systems Netflow Services Export version 9. *RFC 3954* (2004). `https://tools.ietf.org/html/rfc3954`

[48] P4 Language Consortium. 2018. Baseline switch.p4. (2018). `https://github.com/p4lang/switch`

[49] P4 Language Consortium. 2018. Portable Switch Architecture. (2018). `https://p4.org/p4-spec/docs/PSA.html`

[50] James R. Dabrowski and Ethan V. Munson. 2001. Is 100 Milliseconds Too Fast?. In *Proceedings of CHI*.

[51] Jeffrey Dean and Luiz André Barroso. [n. d.]. The Tail at Scale. 56, 2 ([n. d.]).

[52] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of SOSP*.

[53] Linux Networking Documentation. 2022. DCTCP (DataCenter TCP). (2022). `https://www.kernel.org/doc/html/latest/networking/dctcp.html`.

[54] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time Is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review* 36, 1 (2006).

[55] EdgeCore Networks. 2016. AS5900-54X Spec. (2016). `https://bit.ly/2VQ1RZb`

[56] EdgeCore Networks. 2018. AS5812-54X Spec. (2018). `https://goo.gl/ZKqF6F`

[57] EdgeCore Networks. 2019. AS5816-64X Spec. (2019). `https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=166&id=309`

[58] Nick Feamster, Arpit Gupta, Jennifer Rexford, and Walter Willinger. 2019. NSF workshop on measurements for self-driving networks. In *Proceedings of Workshop on Measurements for Self-Driving Networks*.

[59] Nick Feamster and Jennifer Rexford. 2017. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).

[60] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of NSDI*.

[61] Edward John Forrest Jr. 2014. How to Precision Clean All Fiber Optic Connections: A Step By Step Guide. (2014).

[62] Marco Foschiano. 2008. Cisco Systems UniDirectional Link Detection (UDLD) Protocol. IETF RFC 5171. (2008). `https://tools.ietf.org/html/rfc5171`.

[63] The Linux Foundation. 2018. DPDK. (2018). `http://dpdk.org/`

[64] fs.com. [n. d.]. 10GBASE-SR SFP+ optical transceiver. ([n. d.]). `https://bit.ly/3CRJMTK`.

[65] fs.com. [n. d.]. 50GBASE-SR SFP56 optical transceiver. ([n. d.]). `https://bit.ly/3Pb8wuo`.

[66] fs.com. [n. d.]. Edge-Core ET7302-SR compatible 25GBASE-SR optical transceiver. ([n. d.]). `https://bit.ly/3cR3jca`.

[67] fs.com. 2022. QSFP28-SR4-100G Reliability MTBF Test Report. (2022). Retrieved 2022-03-26 from `https://img-en.fs.com/file/report/qsfp28-sr4-100g-reliability-mtbf-test-report.pdf`

[68] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of OSDI*.

[69] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of CoNEXT*.

[70] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When Cloud Storage Meets RDMA. In *Proceedings of NSDI*.

[71] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of SOSP*.

[72] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, et al. 2018. In-network computing to the rescue of faulty links. In *Proceedings of the NetCompute Workshop*.

[73] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of SIGCOMM*.

[74] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *Proceedings of NSDI*.

[75] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of SIGCOMM*.

[76] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *Proceedings of SIGCOMM*.

[77] P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification v1.0. (2018). `https://goo.gl/HtPE9K`

[78] QSFP-DD MSA Group. 2022. QSFP-DD/QSFP-DD800/QSFP112 Hardware Specification. (2022). Retrieved 2022-03-24 from `http://www.qsfp-dd.com/wp-content/uploads/2022/03/QSFP-DD-Hardware-Rev6.2.pdf`

[79] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: a High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of SIGCOMM*.

[80] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of SIGCOMM*.

[81] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. DCell: a Scalable and Fault-tolerant Network structure for Data Centers. In *Procedings of SIGCOMM*.

[82] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and

Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.. In *Proceedings of NSDI*.

[83] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of SIGCOMM*.

[84] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. 2023. Datacenter Ethernet and RDMA: Issues at Hyperscale. *arXiv preprint arXiv:2302.03337* (2023).

[85] Todd Hoff. [n. d.]. Latency Is Everywhere and It Costs You Sales - How to Crush It. ([n. d.]). `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`

[86] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *Proceedings of NSDI*.

[87] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of SIGCOMM*.

[88] IEEE. 2009. 802.11n-2009 Standard. (2009). `https://standards.ieee.org/ieee/802.11n/3952/`.

[89] IEEE. 2013. 802.11ac-2013 Standard. (2013). `https://ieeexplore.ieee.org/document/6687187`.

[90] IEEE. 2015. IEEE Standard for Ethernet - Amendment 3: Physical Layer Specifications and Management Parameters for 40 Gb/s and 100 Gb/s Operation over Fiber Optic Cables. *IEEE Std 802.3bm-2015 (Amendment to IEEE Std 802.3-2012 as amended by IEEE Std 802.3bk-2013 and IEEE Std 802.3bj-2014)* (2015).

[91] IEEE. 2016. IEEE Standard for Ethernet – Amendment 2: Media Access Control

Parameters, Physical Layers, and Management Parameters for 25 Gb/s Operation Amendment 2: Media Access Control Parameters, Physical Layers, and Management Parameters for 25 Gb/s Operation. *IEEE Std 802.3by-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015)* (2016).

[92] IEEE. 2017. IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation. *IEEE Std 802.3bs-2017 (Amendment to IEEE 802.3-2015 as amended by IEEE's 802.3bw-2015, 802.3by-2016, 802.3bq-2016, 802.3bp-2016, 802.3br-2016, 802.3bn-2016, 802.3bz-2016, 802.3bu-2016, 802.3bv-2017, and IEEE 802.3-2015/Cor1-2017)* (2017).

[93] IEEE. 2019. IEEE Standard for Ethernet - Amendment 3: Media Access Control Parameters for 50 Gb/s and Physical Layers and Management Parameters for 50 Gb/s, 100 Gb/s, and 200 Gb/s Operation. *IEEE Std 802.3cd-2018 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018 and IEEE Std 802.3bt-2018)* (2019).

[94] IEEE. 2020. IEEE Standard for Ethernet – Amendment 7: Physical Layer and Management Parameters for 400 Gb/s over Multimode Fiber. *IEEE Std 802.3cm-2020 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018, IEEE Std 802.3bt-2018, IEEE Std 802.3cd-2018, IEEE Std 802.3cn-2019, IEEE Std 802.3cg-2019, and IEEE Std 802.3cq-2020)* (2020).

[95] Intel. 2018. FlexPipe. (2018). `https://goo.gl/PzPudG`

[96] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding up Distributed Request-Response Workflows. In *Proceedings of SIGCOMM*.

[97] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proceedings of SIGCOMM*.

[98] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim,

and David Mazières. 2014. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of SIGCOMM*.

[99] Raj Joshi, Qi Guo, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, and Ben Leong. 2022. LinkGuardian: Mitigating the impact of packet corruption loss with link-local retransmission. In *Proceedings of APNet*.

[100] Raj Joshi, Ben Leong, and Mun Choon Chan. 2019. Timertasks: Towards time-driven execution in programmable dataplanes. In *Proceedings of SIGCOMM (Posters and Demos)*.

[101] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proceedings of NSDI*.

[102] Juniper Networks. 2016. Network Analytics Overview. (2016). `https://goo.gl/TbNwSC`

[103] Zaid Ali Kahn. 2016. Project Falco: Decoupling Switching Hardware and Software. (2016). `https://goo.gl/U7PUQZ`

[104] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic Load Balancing Without Packet Reordering. *SIGCOMM CCR* 37, 2 (2007), 51–62.

[105] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. 2021. Debugging Transient Faults in Data Centers using Synchronized Network-wide Packet Histories. In *Proceedings of NSDI*.

[106] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. 2019. Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs. In *Proceedings of SOSR*.

[107] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. 2013. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of CoNext*.

[108] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High

Bandwidth-Delay Product Networks. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002).

[109] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of SOSR*.

[110] Changhoon Kim and Roberto Mari. 2018. Advanced Dataplane Telemetry. (2018). = https://opennetworking.org/wp-content/uploads/2018/12/Data-Plane-Telemetry-ONF-Connect-Public.pdf,.

[111] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *Proceedings of SIGCOMM (Poster)*.

[112] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *Proceedings of HotNets*.

[113] Ron Kohavi and Roger Longbotham. [n. d.]. Online Experiments: Lessons Learned. 40, 9 ([n. d.]). `http://ieeexplore.ieee.org/document/4302627/`

[114] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. [n. d.]. Controlled Experiments on the Web: Survey and Practical Guide. 18, 1 ([n. d.]).

[115] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of SIGCOMM*.

[116] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving convergence-free routing using failure-carrying packets. In *Proceedings of SIGCOMM*.

[117] Jeongkeun Lee. 2020. Advanced Congestion & Flow Control with Programmable Switches. In *P4 Expert Roundtable Series*. `https://bit.ly/3J8x7fw`

[118] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of SoCC*.

[119] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of SIGCOMM*.

[120] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings EuroSys*.

[121] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *Proceedings of NSDI*.

[122] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E Anderson. 2013. F10: A Fault-Tolerant Engineered Network.. In *Procedings of NSDI*.

[123] Suksant Sae Lor, Raul Landa, and Miguel Rio. 2010. Packet Re-cycling: Eliminating Packet Losses due to Network Failures. In *Proceedings of HotNets*.

[124] Markets and Markets. 2021. Cloud Computing Market Forecast. (2021). Retrieved 2022-03-24 from `https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html`

[125] Richard Martin. 2007. Wall Street's Quest To Process Data At The Speed Of Light. Information Week. (2007).

[126] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient scheduling policies for {Microsecond-Scale} tasks. In *Proceedings of NSDI*.

[127] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onu Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *IMC*.

[128] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large

Scale Study of Data Center Network Reliability. In *Proceedings of the Internet Measurement Conference*.

[129] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of SIGCOMM*.

[130] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of SIGCOMM*.

[131] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of SIGCOMM*.

[132] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of SIGCOMM*.

[133] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2022. FAst In-Network *GraY* Failure Detection for ISPs. In *Proceedings of SIGCOMM*.

[134] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of SIGCOMM*.

[135] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. In *Proceedings of SIGCOMM*.

[136] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. 2015. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proceedings of SIGCOMM*.

[137] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017.

Language-directed hardware design for network performance monitoring. In *Proceedings of SIGCOMM*.

[138] Arista Networks. 2015. Latency Analyzer (LANZ) Architectures and Configuration. (2015). https://goo.gl/LrRNi4

[139] Barefoot Networks. 2018. Tofino. (2018). https://goo.gl/cdEK1E

[140] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Procedings of SIGCOMM*.

[141] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of SIGCOMM*.

[142] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of NSDI*.

[143] Ntop. 2018. n2disk. (2018). https://goo.gl/7DFkSp

[144] NVIDIA. 2022. Mellanox Connect X-6. (2022). https://www.nvidia.com/en-sg/networking/ethernet/connectx-6.

[145] NVIDIA. 2022. RDMA Transport Modes. (2022). https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17/Transport+Modes.

[146] NVIDIA. 2022. RoCE Selective Repeat. (2022). https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=25137694.

[147] Ping Pan, George Swallow, and Alia Atlas. 2005. Fast reroute extensions to RSVP-TE for LSP tunnels. IETF RFC 4090. (2005). https://tools.ietf.org/html/rfc4090.

[148] Christina Parsa and JJ Garcia-Luna-Aceves. 1999. TULIP: A Link-Level Protocol for Improving TCP over Wireless Links. In *Proceedings of WCNC*.

[149] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of SIGCOMM*.

[150] Peter Phaal. 2004. sFlow. (2004). `http://sflow.org/sflow`

[151] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. 2008. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proceedings of FAST*.

[152] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. 2019. SQR: In-network packet loss recovery from link failures for highly reliable datacenter networks. In *Proceedings of ICNP*.

[153] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: Congestion Signals Are Simple and Effective for Network Load Balancing. In *Proceedings of SIGCOMM*.

[154] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's datacenter network. In *Proceedings of SIGCOMM*.

[155] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's Time for Low Latency. In *Proceedings of HotOS*.

[156] Matt Sargent, Mark Allman, and Vern Paxson. 2011. Computing TCP's Retransmission Timer. IETF RFC 6298. (2011). `https://tools.ietf.org/html/rfc6298`.

[157] Eric Schurman and Jake Brutlag. 2009. The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search. (2009).

[158] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting Emerging Applications With Low-Latency Failover in P4. In *Proceedings of NEAT*.

[159] Seladb. 2018. PcapPlusPlus. (2018). `https://github.com/seladb/PcapPlusPlus`

[160] Danfeng Shan, Fengyuan Ren, Peng Cheng, Ran Shu, and Chuanxiong Guo. 2018. Micro-Burst in Data Centers: Observations, Analysis, and Mitigations. In *Proceedings of ICNP*.

[161] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *Proceedings of NSDI*.

[162] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. 2013. OpenFlow: Meeting carrier-grade recovery requirements. *Computer Communications* 36, 6 (2013), 656–665.

[163] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *Proceedings of NSDI*.

[164] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Proceedings of SIGCOMM*.

[165] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. 2012. Jellyfish: Networking data centers randomly. In *Proceedings of NSDI*.

[166] R Sivaram. 2008. Some Measured Google Flow Sizes. *Google internal memo, available on request* (2008).

[167] SNIA. 2018. SFF-8679: QSFP+ 4X Hardware and Electrical Specification. (2018). `https://members.snia.org/document/dl/25969`

[168] Steve Sounders. 2009. Velocity and the Bottom Line. (2009). Retrieved 2022-04-22 from http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html

[169] Statista. 2021. Worldwide Internet Usage. (2021). Retrieved 2022-03-24 from https://www.statista.com/statistics/617136/digital-population-worldwide/

[170] TechCrunch. 2013. Microsoft To Refund Windows Azure Customers Hit By 12 Hour Outage That Disrupted Xbox Live. (2013). Retrieved 2022-03-25 from https://techcrunch.com/2013/02/24/microsoft-to-refund-windows-azure-customers-hit-by-12-hour-outage-that-disrupted-xbox-live/

[171] Frank Uyeda, Luca Foschini, Fred Baker, Subhash Suri, and George Varghese. 2011. Efficiently Measuring Bandwidth at All Time Scales. In *Proceedings of NSDI*.

[172] Amin Vahdat. 2017. ONS Keynote: Cloud Native Networking. (2017). https://youtu.be/1xBZ5DGZZmQ?t=1460

[173] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. 2012. Deadline-aware datacenter TCP (D2TCP). In *Proceedings of SIGCOMM*.

[174] Niels LM Van Adrichem, Benjamin J Van Asten, and Fernando A Kuipers. 2014. Fast recovery in software-defined networks. In *Proceedings of EWSDN*.

[175] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proceedings of SIGCOMM*.

[176] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. 2012. More is Less: Reducing Latency via Redundancy. In *Proceedings of HotNets*.

[177] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai,

Hongqiang Harry Liu, and Ming Zhang. 2022. Predictable vFabric on Informative Data Plane. In *Proceedings of SIGCOMM*.

[178] Jim Warner. [n. d.]. Packet Buffers. ([n. d.]). `https://people.ucsc.edu/~warner/buffer.html`.

[179] Jim Warner. 2019. Packet Buffers. (2019). `https://people.ucsc.edu/~warner/buffer.html`

[180] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of SIGCOMM*.

[181] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. 2018. Masking Failures from Application Performance in Data Center Networks with Shareable Backup. In *Procedings of SIGCOMM*.

[182] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of SIGCOMM*.

[183] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using Trio: Juniper Networks' programmable chipset-for emerging in-network applications. In *Proceedings of SIGCOMM*.

[184] Kiran Yedugundla, Per Hurtig, and Anna Brunstrom. 2017. Probe or Wait: Handling tail losses using Multipath TCP. In *Proceddings of IFIP Networking*.

[185] David Zats, Anand Padmanabha Iyer, Randy H. Katz, Ion Stoica, and Amin Vahdat. 2013. FastLane: An Agile Congestion Signaling Mechanism for Improving Datacenter Performance. In *Proceedings of SoCC*.

[186] Gaoxiong Zeng, Li Chen, Bairen Yi, and Kai Chen. 2022. Cutting Tail Latency in Commodity Datacenters with Cloudburst. In *Proceedings of INFOCOM*.

[187] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *Proceedings of FAST*.

[188] Qiao Zhang, Vincent Liu, and Hongyi Zeng. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of IMC*.

[189] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow event telemetry on programmable data plane. In *Proceedings of SIGCOMM*.

[190] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proceedings of SIGCOMM*.

[191] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*.

[192] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of SIGCOMM*.

[193] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. 2017. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *Proceedings of NSDI*.